Karlsruhe Institute of Technology

Software Engineering II

Prof. Dr. Raffaela Mirandola

Topic 15

Software Quality – Reliability

SASIS – SELF-ADAPTIVE SOFTWARE-INTENSIVE SYSTEMS
KASTEL – INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY

sasis.kastel.kit.edu

Overview on this Lecture

Content

- Dimensions of Software Quality (in particular Reliability)
- Real-time system basics
- Types of real-time systems and examples
- Safety and Reliability Patterns for RT-Design

Learning Goals, participants are able –

- to know and distinguish different kinds of software qualities
- to explain the concept of a real-time system
  - and why these systems are usually implemented as concurrent processes
- to distinguish between different types of real-time systems
  - monitoring and control systems and data acquisition systems
- to select a safety and reliability pattern for a given problem

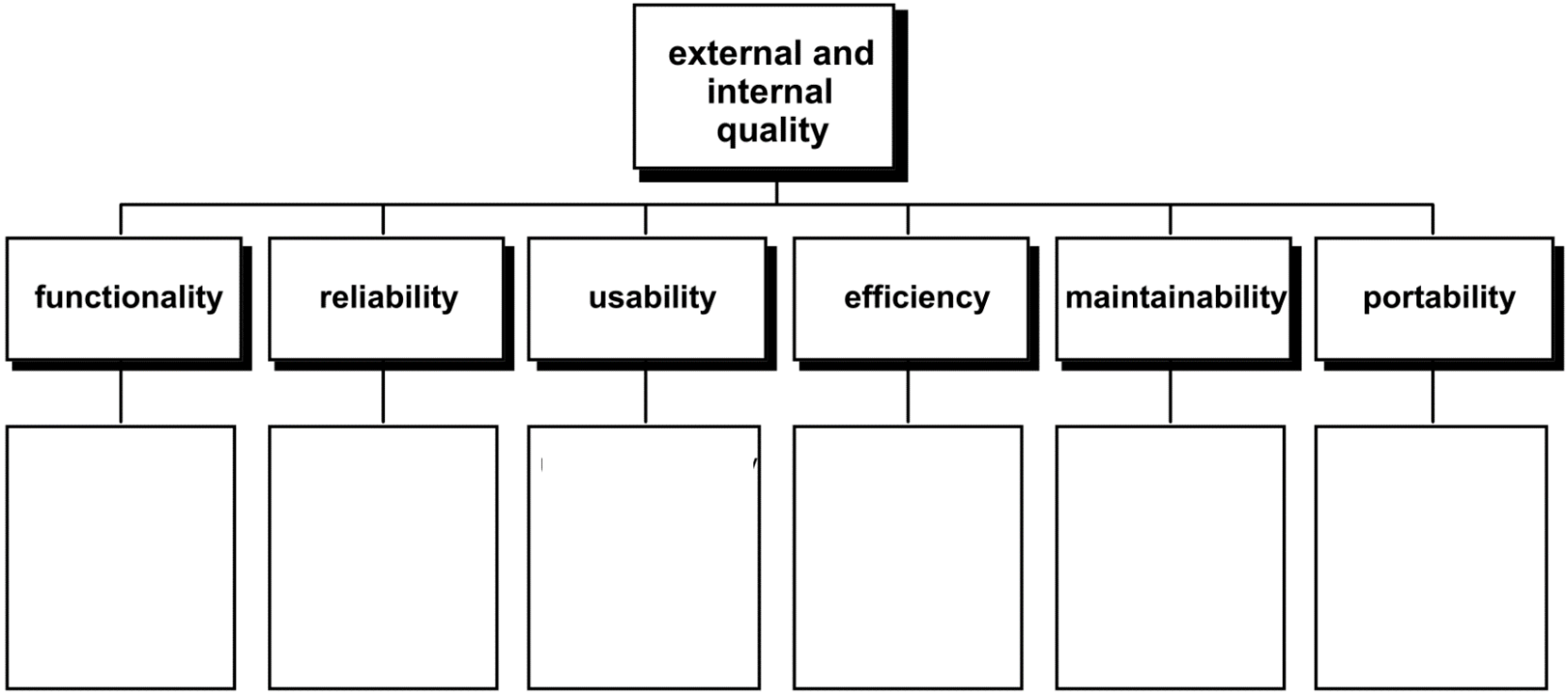
most slides taken from [Sommerville], patterns taken from [Douglass 2003]

SOFTWARE QUALITY

Dimensions of Software Quality

- *"The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders … stakeholders' needs (functionality, performance, security, maintainability, etc.) are precisely what is represented in the quality model …"*

– [ISO/IEC 25010:2011]

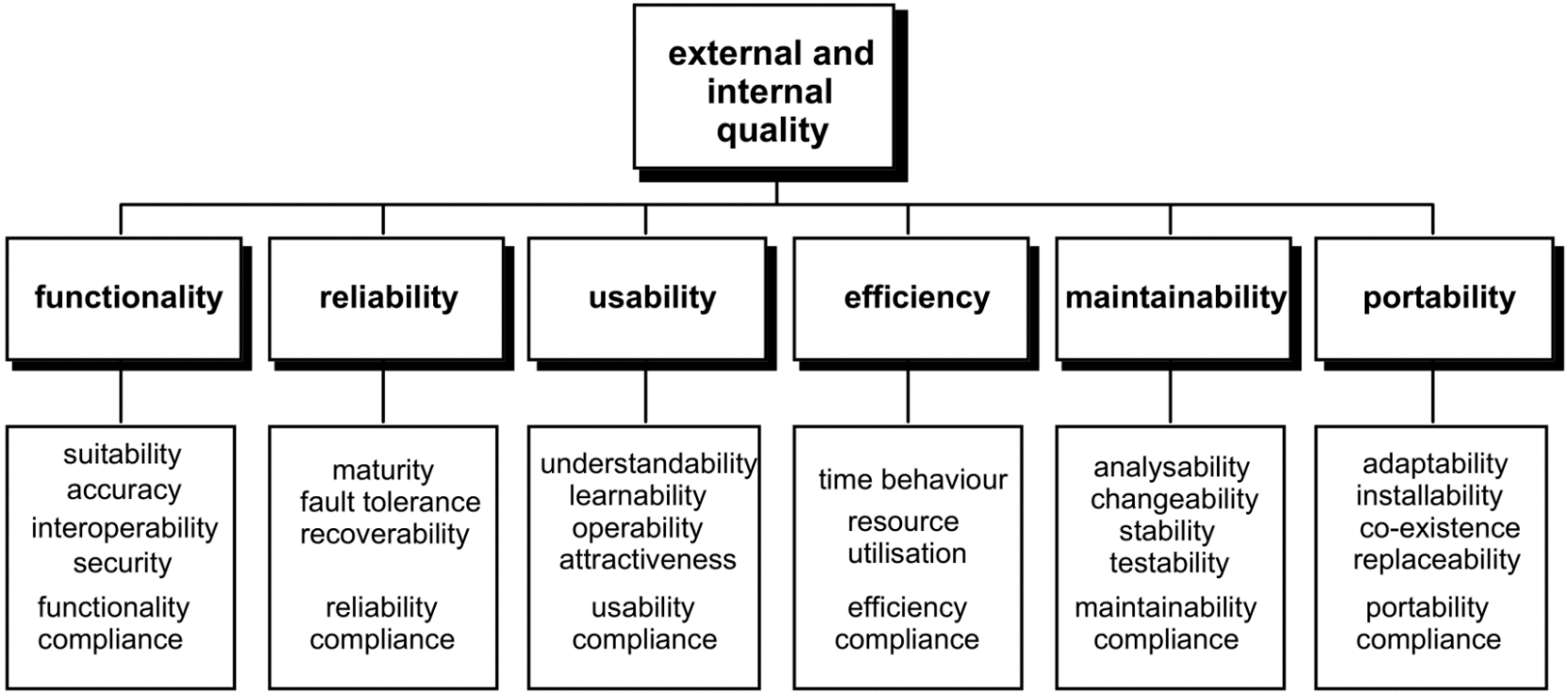


[ISO/IEC 25010:2011]

Dimensions of Software Quality

- *"The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders … stakeholders' needs (functionality, performance, security, maintainability, etc.) are precisely what is represented in the quality model …"*

– [ISO/IEC 25010:2011]

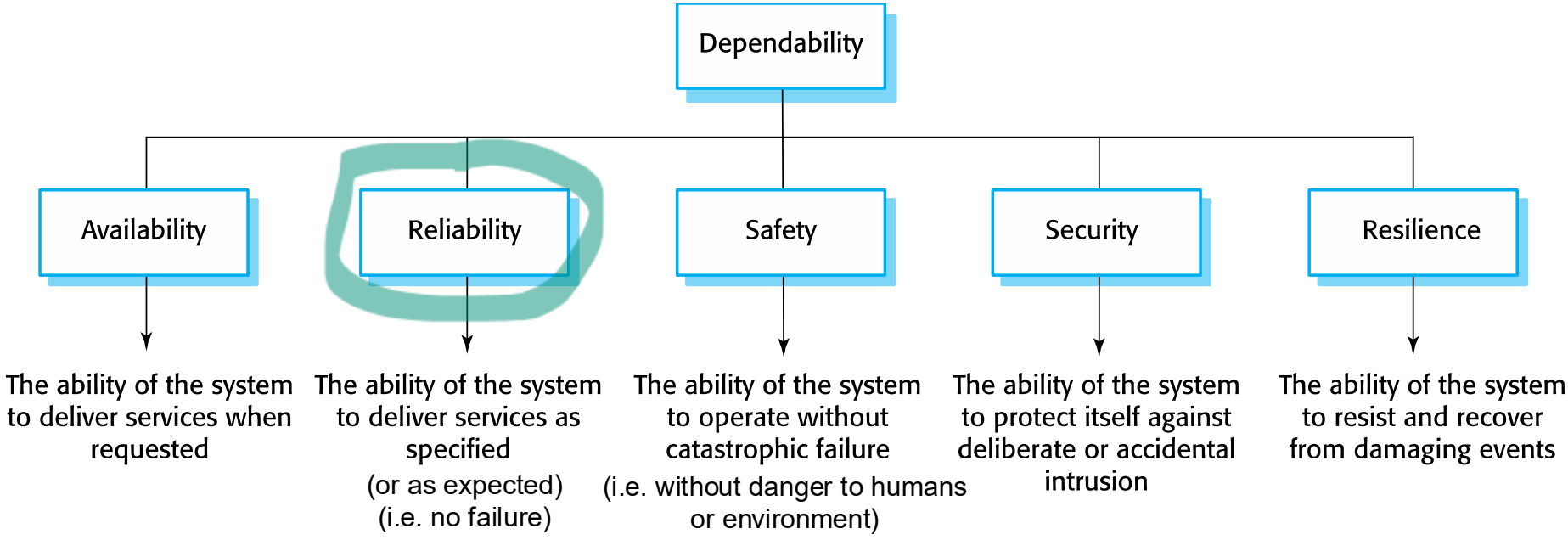


[ISO/IEC 25010:2011]

Quality Metrics

- Performance
  - Response time
  - Throughput
- Reliability
  - Probability of failure on demand (POFOD)
  - Mean time to failure (MTTF)
- Security
  - Mean time to break component (MTTBC)
  - Mean time to security failure (MTTSF)

Dimensions of Dependability



[Sommerville 2017]

REAL-TIME SYSTEMS

Real-time Systems

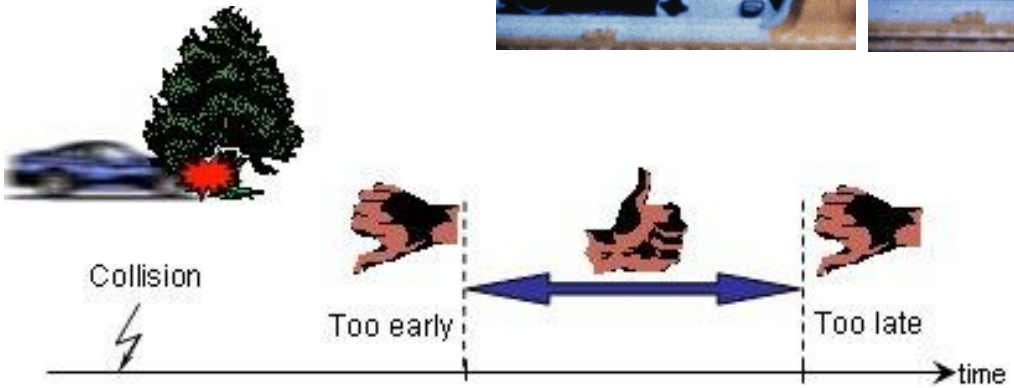
- Mostly systems which monitor and control their environment
  - ➔ usually *embedded systems*
    - but not all embedded systems are real-time systems
- Usually associated with hardware devices
  - sensors: collect data from the system environment
  - actuators: change (in some way) the system's environment

- Time is critical!
  - ➔ real-time systems MUST respond within specified times
  - typically they must process data at least as quick as it arrives
    - however, if data collection is faster than processing data must be buffered
      - e.g. when collecting information about particle collisions at LHC@CERN
    - circular or ring buffers are a mechanism for smoothing out small speed differences

Logical vs. Temporal Correctness

- The result of (hard) real-time data processing is correct, iff
  - logically correct AND
  - temporally correct

- Example: Airbag must be fully inflated within a certain interval



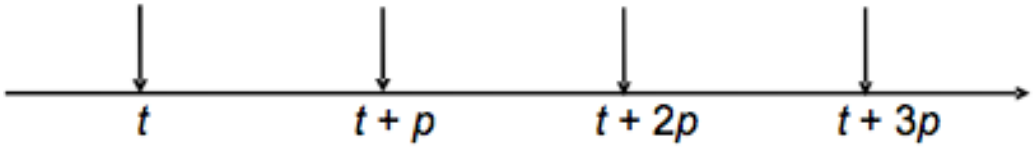
Some Definitions

- Real-time system is a software system where the correct functioning of the system depends on the results produced by the system
  - AND the time at which these results are produced
- Soft real-time system is a system whose operation is degraded if results are not produced according to the specified timing constraints
- Hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specification
  - ➔ *Accordingly people often talk about soft resp. hard real-time requirements*
- **Any other use of "*real-time*" (in particular in enterprise systems) means "*online*"**
- System response times must be considered at design-time
  - Thus, real-time systems are often defined by listing the possible stimuli, the expected responses and the timing constraints

Stimulus/Response Systems

Given a *stimulus*, the system must produce a *response* within a specified time frame

- two general cases –
  - periodic stimuli occur at predictable time intervals
    Example: a temperature sensor is polled 10 times per second



  - aperiodic stimuli occur at unpredictable times
    Example: a system power failure may trigger an interrupt, which must be processed by the system

➔ *if the processor is powerful enough, interrupts can be replaced with periodic polling*

Interrupts vs. Periodic Processes

Interrupts

- control is automatically transferred to a pre-determined memory location
  - containing an instruction to jump to an interrupt service routine
- further interrupts are disabled
- when the interrupt is serviced, control is returned to interrupted process

➔ *interrupt service routines MUST be short, simple and fast*

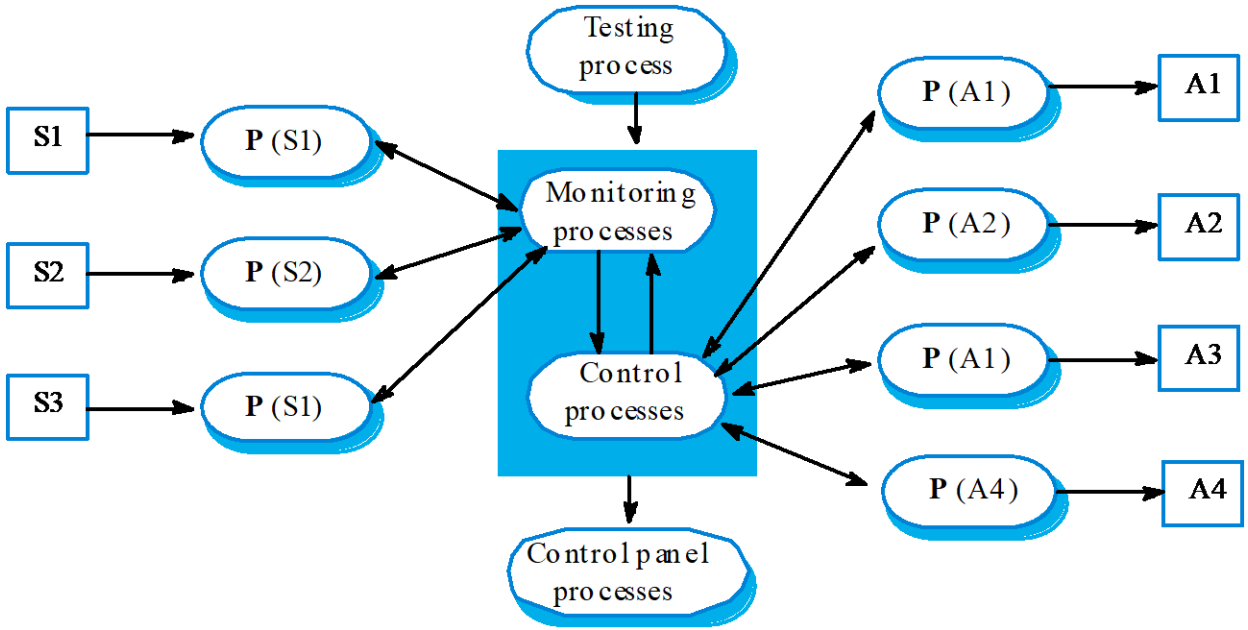
Most real-time systems encompass several classes of periodic processes

- each with different period, execution time, and deadline
- the real-time clock ticks periodically
  - each tick causes interrupt which triggers management of periodic processes
  - dispatcher selects a process which is ready for execution
    - if two processes are due to start, usually the one with the shorter deadline goes first

Monitoring & Control Systems

… are important classes of real-time systems

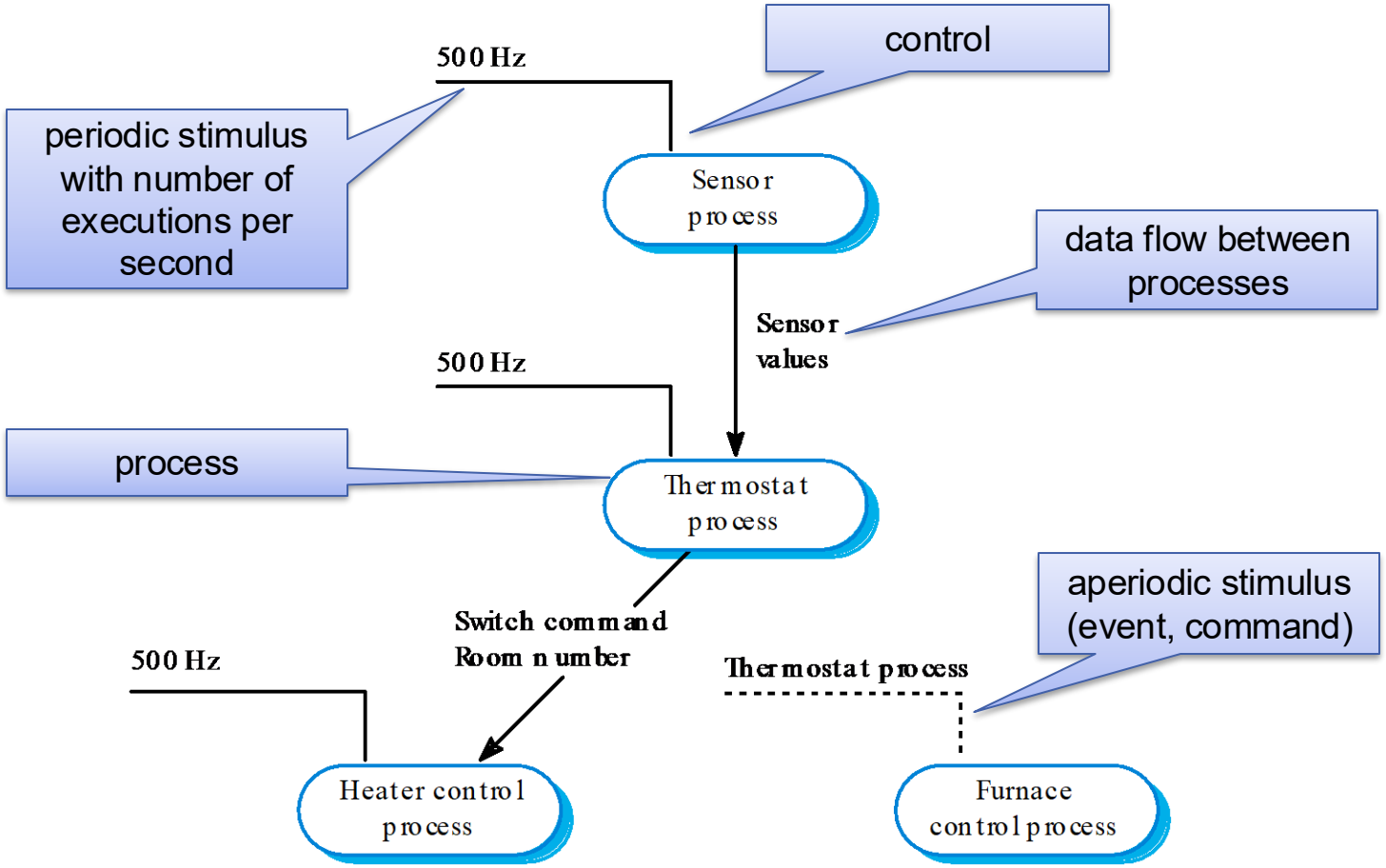
- continuously check sensors and take actions depending on sensor values
- **Monitoring** systems act when exceptional sensor values are detected
- **Control** systems continuously control hardware actuators depending on value of associated sensor



[Sommerville]

Example Monitoring and Control Systems

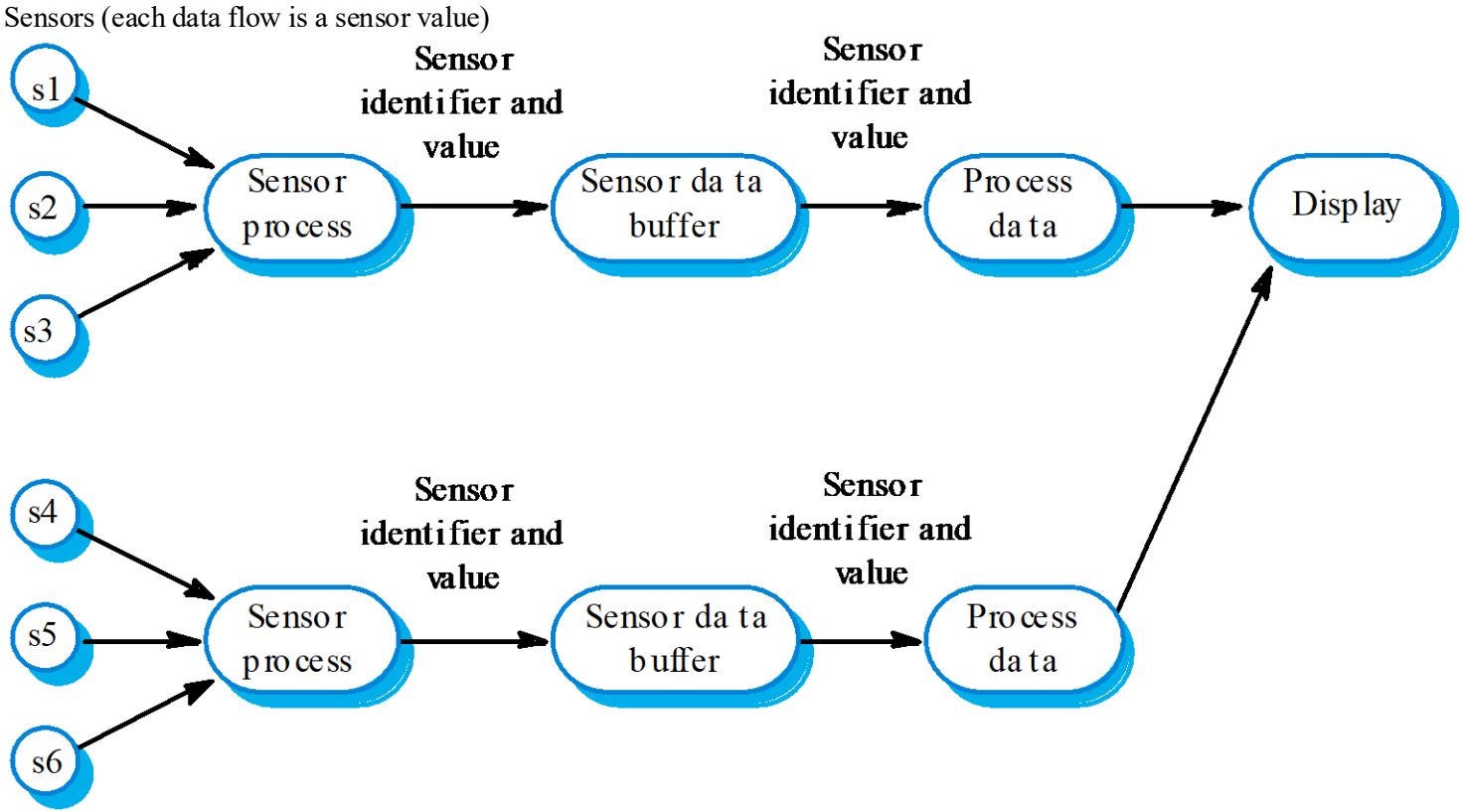
- *Heating control system* monitors temperature and turns heaters on and off



Data Acquisition Systems

… as a third class of RT systems

- collect data from sensors for subsequent processing and analysis
  - where data collection processes and processing processes may have different periods and deadlines

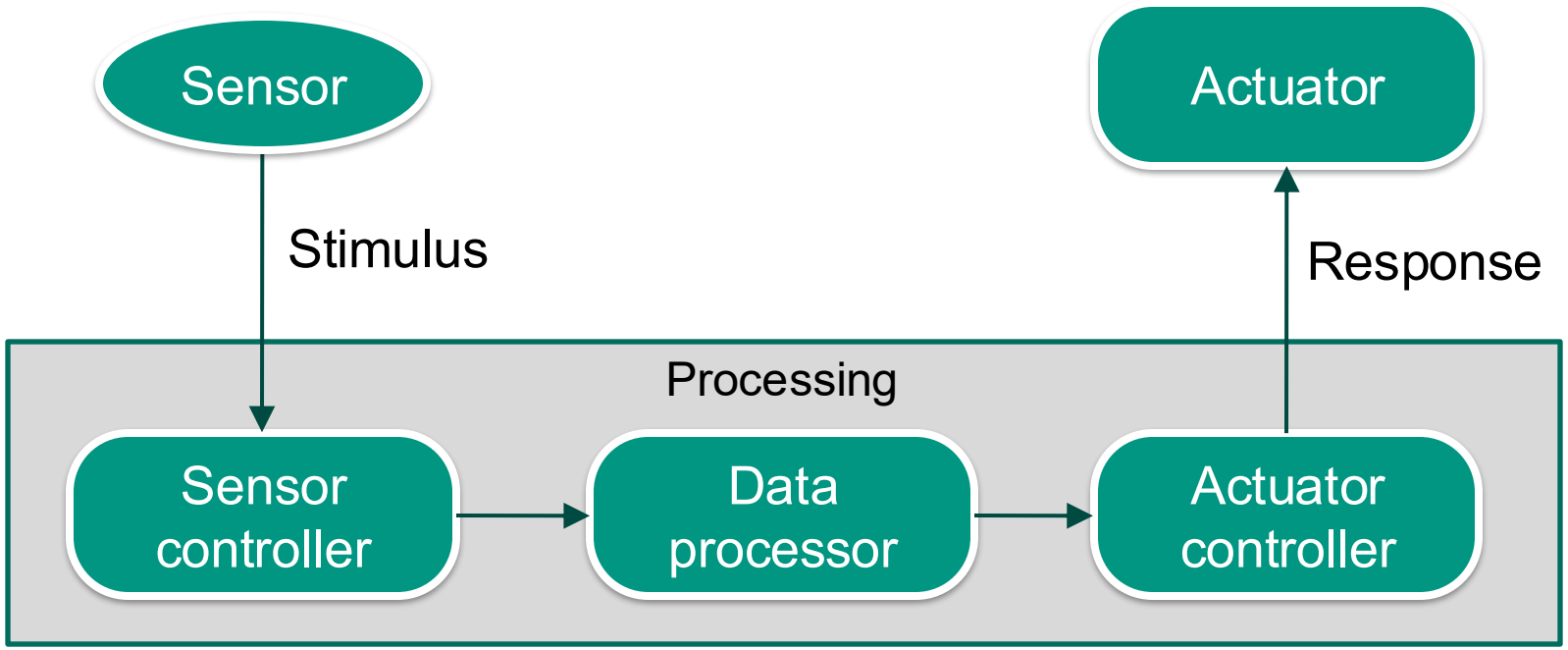


The Large Hadron Collider will produce roughly 15 petabytes (15 million gigabytes) of data annually – enough to fill more than 1.7 million dual-layer DVDs a year!

[CERN]

Basic Sensor/Actuator Schema

- Sensors generate stimuli to be processed by system
- System generates and sends response to an actuator



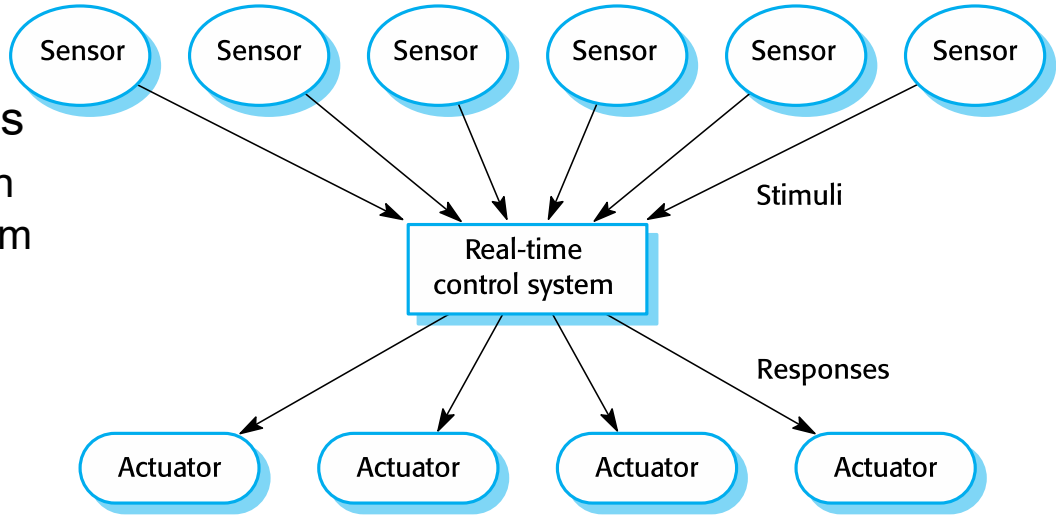
➔ If sensor and actuator deal with same physical parameters, this becomes a control loop

- *designing the processing requires knowledge from control engineering*

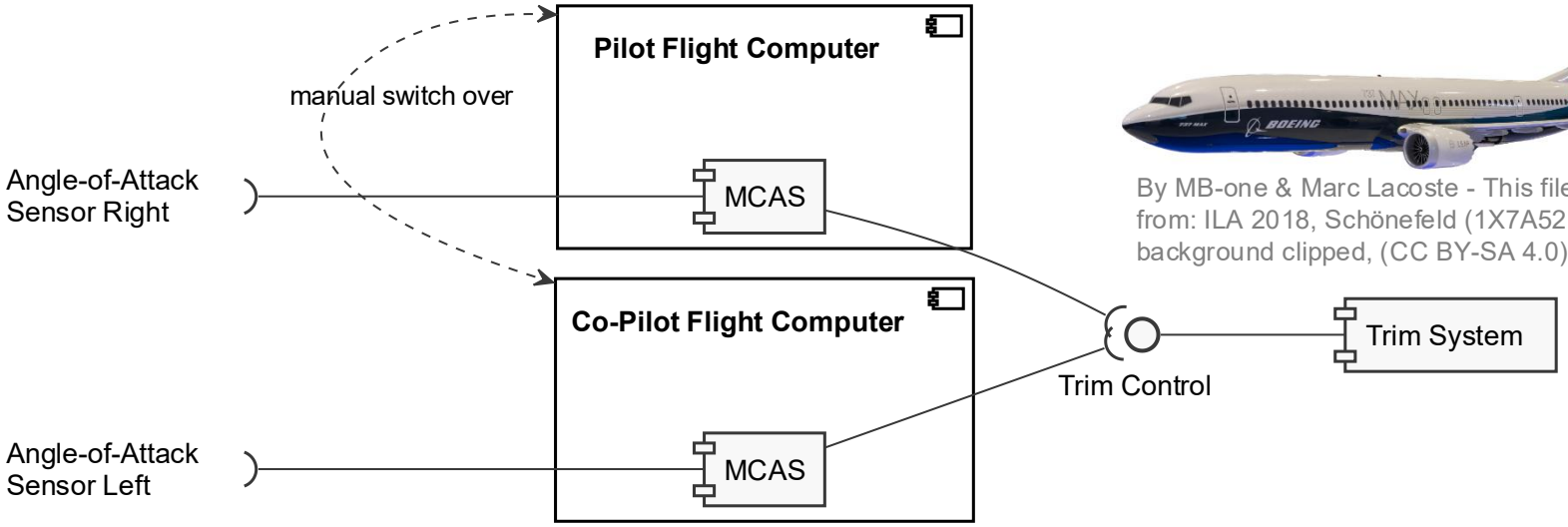
Design Considerations

- Because of timing demands made by different stimuli/responses –
  - every sensor/actor pair should have its own process
  - system architecture must allow for fast switching between different stimulus handlers

- Timing demands of different stimuli may be different
  - ➔ simple sequential loops are usually not adequate to implement controllers
  - real-time systems are therefore designed as cooperating processes
    - with real-time execution platform controlling them



Example
Maneuvering Characteristics Augmentation System (MCAS)



By MB-one & Marc Lacoste - This file was derived from: ILA 2018, Schönefeld (1X7A5288).jpg with background clipped, (CC BY-SA 4.0)

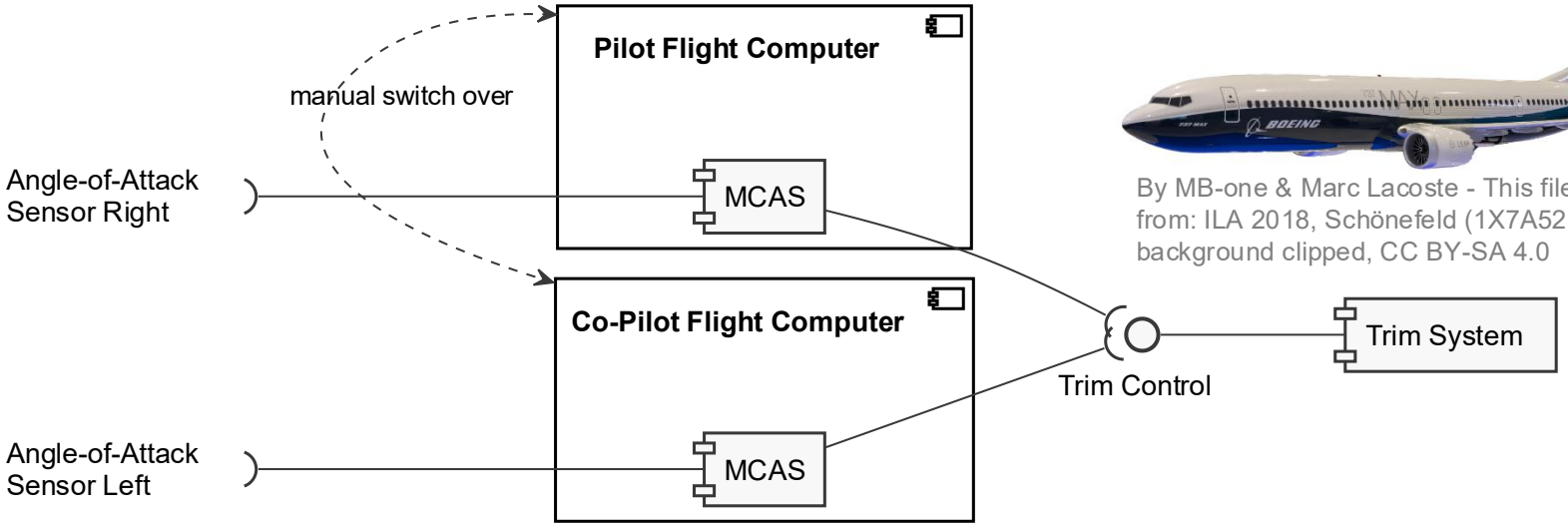
MCAS reduces aircraft's pitch, if angle of attack exceeds threshold at which aircraft might stall and thus crash.

- MCAS evaluates angle-of-attack and, if limit exceeded, sends control signal to trim system within 100 milliseconds
- One MCAS and Angle-of-Attack Sensor per flight computer (pilot and co-pilot)
- Only one flight computer is active
- Requires manual switch over between flight computers

Is MCAS a monitoring or a control system?

Is MCAS a hard or soft real-time system?

Example
Maneuvering Characteristics Augmentation System (MCAS)

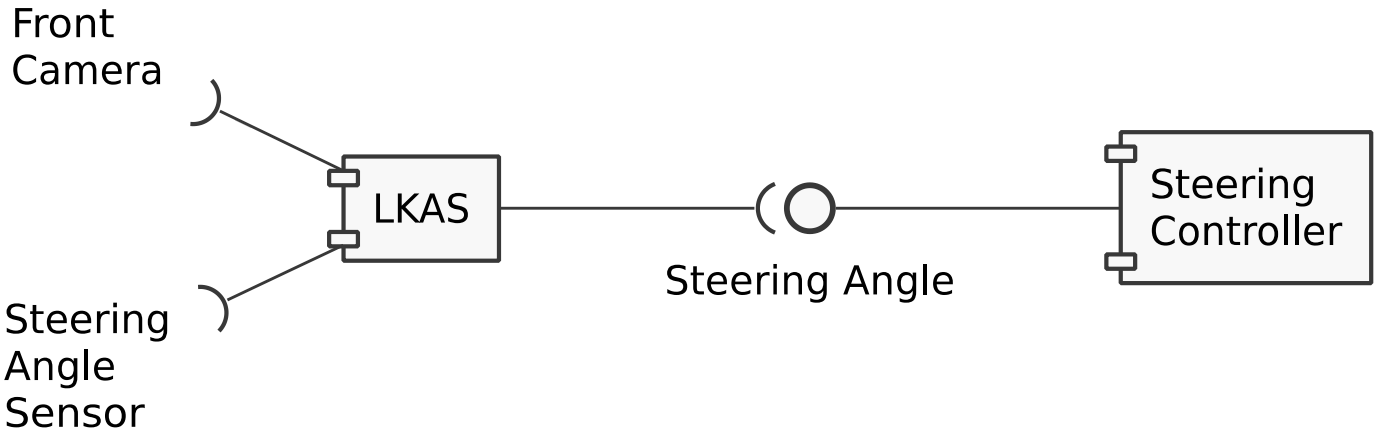


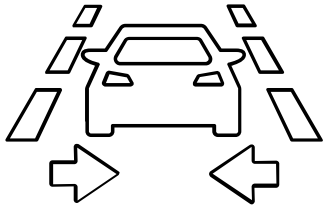
By MB-one & Marc Lacoste - This file was derived from: ILA 2018, Schönefeld (1X7A5288).jpg with background clipped, CC BY-SA 4.0

MCAS reduces aircraft's pitch, if angle of attack exceeds threshold at which aircraft might stall and thus crash.

- **Monitoring system**, as MCAS
  - Continuously monitors angel-of-attack position
  - Only sends control signal, if threshold exceeded
- **Hard real-time system**, as violation of deadline does
  - entail damage to material
  - entail human casualties

Example
Lane Keep Assistent System (LKAS)





Ika car lane keep assist
by Gianluca Pettenon
from the Noun Project
(CC-BY-SA3.0)

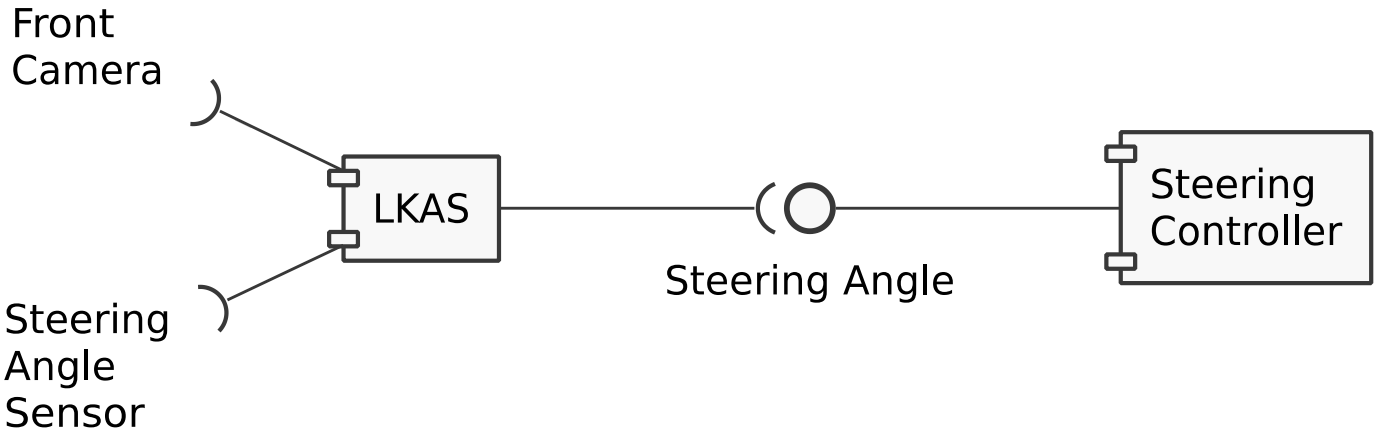
LKAS steers vehicle and keeps it in the middle of the lane

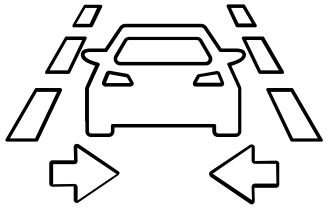
- *Front Camera* continuously determines current position within lane
- *LKAS* determines optimal steering angle and
- sends it to *Steering Controller* every 100ms.
- if deadline exceeded, no steering angle is sent,
  but a new calculation started based on new position
- Driver **must** continuously monitor street to take over steering if necessary

Is LKAS a *monitoring* or a *control* system?

Is LKAS a *hard* or *soft* real-time system?

Example
Lane Keep Assistant System (LKAS)





lka car lane keep assist
by Gianluca Pettenon
from the Noun Project
(CC-BY-SA3.0)

LKAS steers vehicle and keeps it in the middle of the lane

- **Control system**, as LKAS
  - Continuously monitors car's position
  - Periodically sends optimal steering angle to Steering Controller
- **Soft real-time system**, as violation of deadline does
  - not entail damage
  - only entails degradation of quality

Safety & Reliability

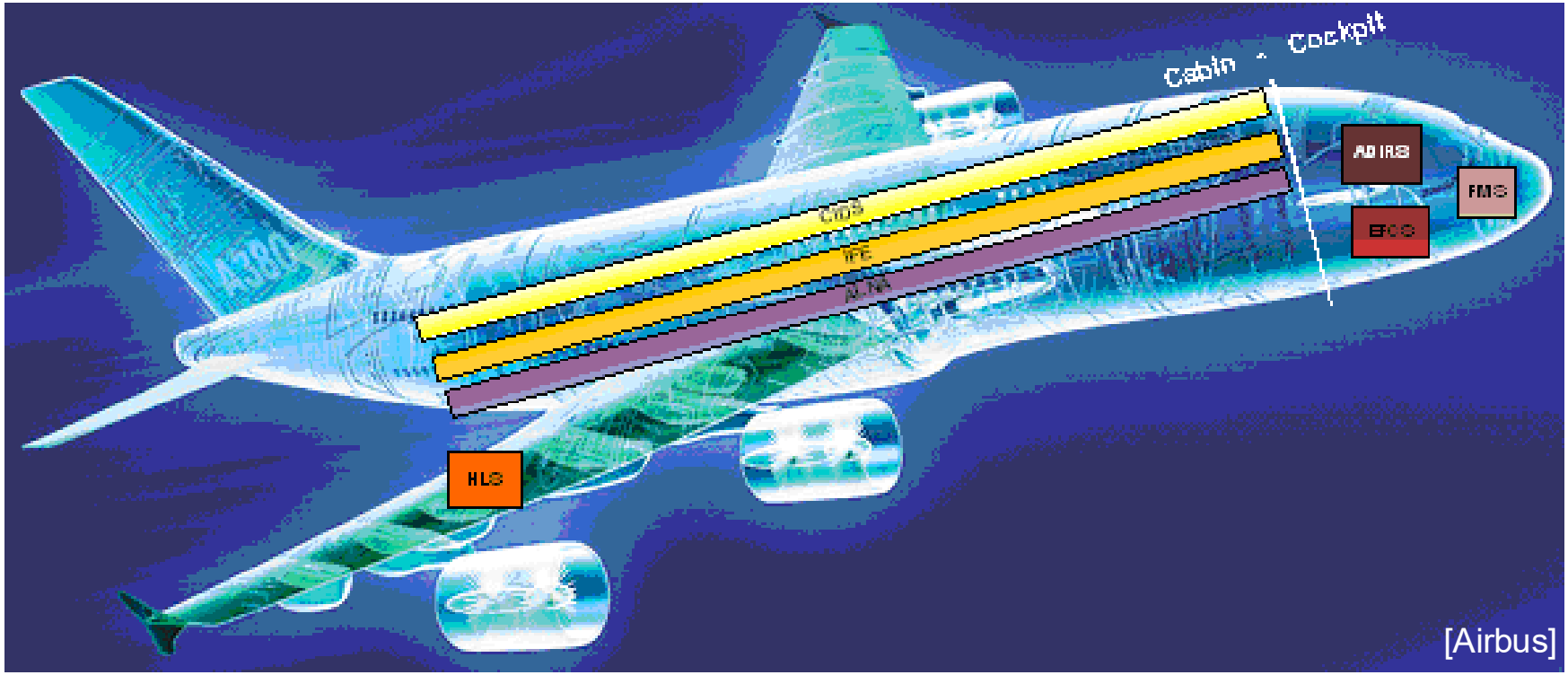
- Safety: Absence of danger for humans and environment
  - absence of catastrophic failure
  - do not confuse with security
- Reliability: probability or duration of failure-free operation
  - whether failures result in danger depends on system's deployment environment
  - in embedded systems, reliability often increase by means of redundancy
- Safety != Reliability
  - safe systems may fail frequently as long as they do not cause accidents,
  - reliability of system does not refer at all to consequences of failures, if they should occur.
  - **fail-safe state** is a system state where there is no likelihood of danger (for the environment)
    - a system in this state often does not operate ☺
    - some *"not so"* critical systems may enter this state after detecting faults
  - ➔ If no fail-safe state exists in a safety-critical system, an increase of reliability tends to increase safety

Notabene: Reliability != Correctness

- Remember:
  - Correctness: Refinement relation between a formal specification and a realisation (design or code)
  - Reliability: Probability of expected behaviour under specified context conditions (usage-profile, execution environment, external services)

- Can a system be reliable, but not correct?
  - Yes: bug is present, but only in code not used in this usage-profile
- Can a system be correct, but not reliable?
  - Yes: specification is incorrect

Example Avionics



- The A380 contains numerous safety critical systems
  - see e.g. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6401111 for some more information

Fault-Error-Failure Chain

Fault

- defect in a system
- may or may not lead to a failure.
- *For instance, although a system may contain a fault, its input and state conditions may never cause this fault to be executed, such that an error occurs; and thus that particular fault never leads to a failure.*
- is usually referred to as a bug for historic reasons

Error

- discrepancy between intended behaviour of system and actual behaviour inside system boundary
- occurs at runtime when some part of the system enters unexpected state due to activation of fault

Failure

- instance in time when system shows behaviour contrary to its specification [or user expectations]

*"An **error** may not necessarily cause a **failure**, for instance, an exception may be thrown by a system, but this may be caught and handled using **fault tolerance** techniques, such that the overall operation of the system conforms to the specification."*

From https://en.wikipedia.org/wiki/Dependability#Threats

Types of Faults

Systematic vs. Random Faults

- Systematic faults (also "design faults“) are mistakes made at either design or build time.
- Random faults occur because something that at one time worked is now broken.
  - Transient faults disappear after a while: Retry helps
  - Persistent faults stay until intervention: E.g. broken hardware

REAL-TIME PATTERNS

Real-Time Patterns

A design pattern is a solution to recurring software design problems

➔ RT-pattern is a solution to recurring software design problems in the area of real-time systems

In the following –

- overview on safety and reliability patterns
  - other patterns, such as, RT distribution or resource patterns cannot be discussed in detail

More details can be found in –

- B.P. Douglass, Real-Time design patterns, Addison-Wesley, 2003
- from where (most of) the following material originates

Architectural Pattern: Channel

- A *channel* can be thought of as a pipe* that sequentially transforms data from an input value to an output value.
  - architectural pattern for RT systems
  - internal elements work on data stream in a kind of factory automation process.
  - each internal element performs relatively simple operations on data
  - multiple elements of the data stream can be in different parts of the channel at the same time.
- Multiple channels to improve quality
  - Performance: throughput capacity can be increased by adding multiple homogeneous (identical) channels.
  - Reliability: multiple channels can operate in various ways to achieve fault tolerance.
  - Safety: multiple channels can improve safety by adding fault identification and safety measures.

*) this pattern is also called the *Pipe and Filter* pattern

Safety & Reliability Pattern Overview

[Douglass 2003, Cha. 9]

Without Fail-Safe State

- Homogeneous Redundancy *(Cha. 9.3)*
- Triple Modular Redundancy *(Cha. 9.4)*
- Heterogeneous Redundancy *(Cha. 9.5)*

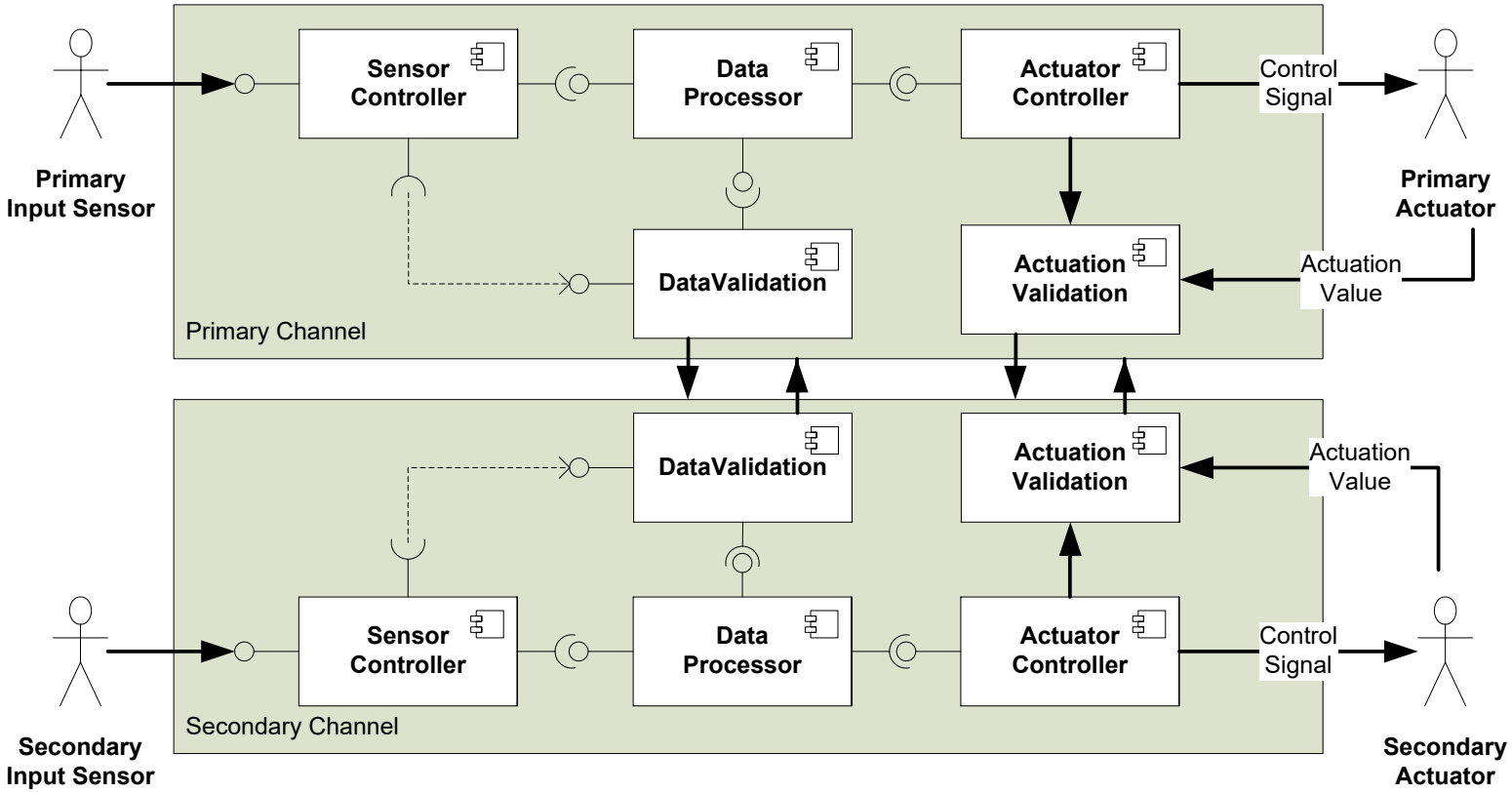
With Fail-Safe State

- Protected Single Channel Pattern *(Cha. 9.2)*
- Monitor-Actuator Pattern *(Cha. 9.6)*
- Sanity Check Pattern *(Cha. 9.7)*
- Watchdog Pattern *(Cha. 9.8)*
- Safety Executive Pattern *(Cha. 9.9)*

Homogeneous Redundancy *(Switch to Backup)*

[Douglass 2003, Cha. 9.3]

- Protection against random faults, not assuming a fail-safe state
  - ➔ a backup channel is required

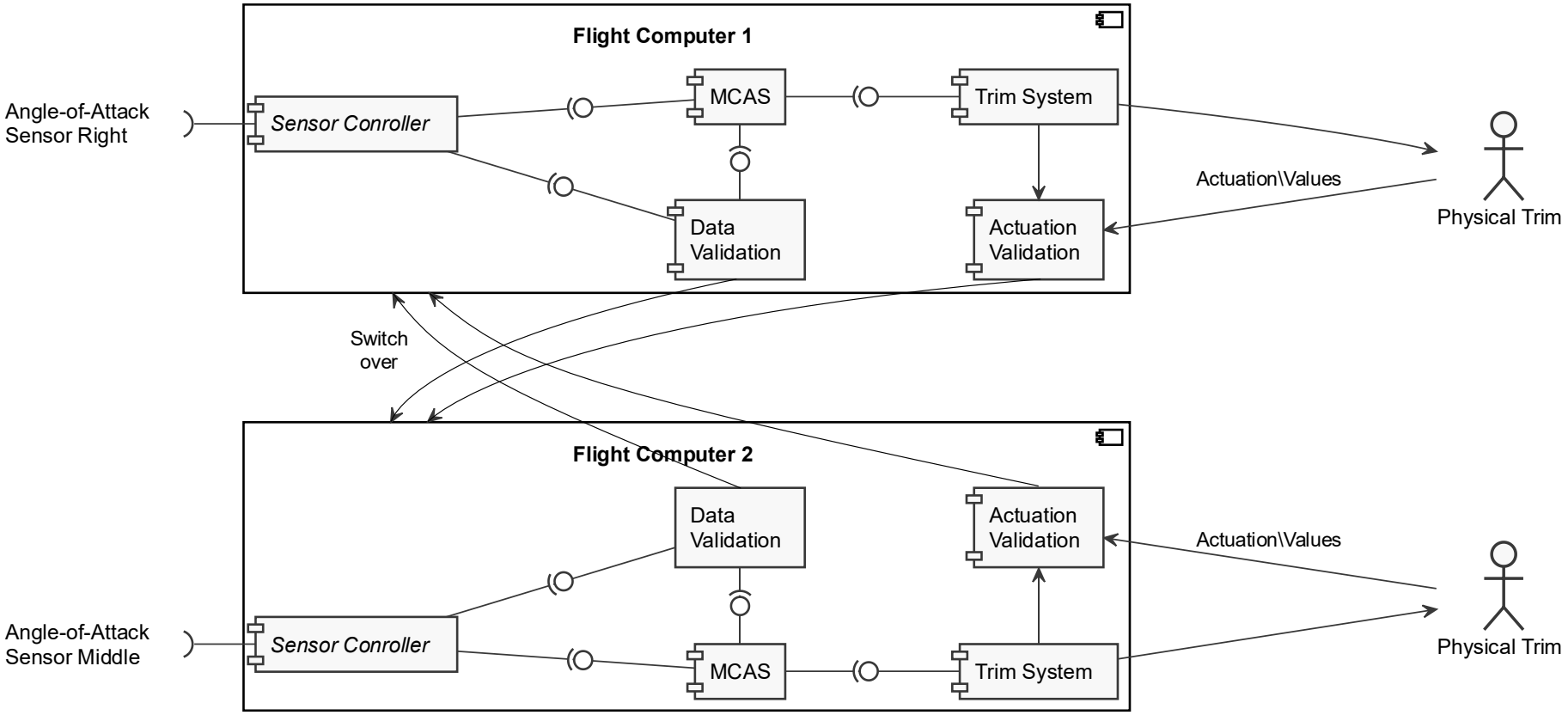


Protected Single Channel, Homogeneous Redundancy, and Heterogeneous Redundancy (later) can have Data Validation and Actuation Validation or just one of the two

Example: Homogeneous Redundancy

- Flight Computers connected to seperate sensors / actuators
- Automatically switches over in case of fault

By MB-one & Marc Lacoste - This file was derived from: ILA 2018, Schönefeld (1X7A5288).jpg with background clipped, CC BY-SA 4.0

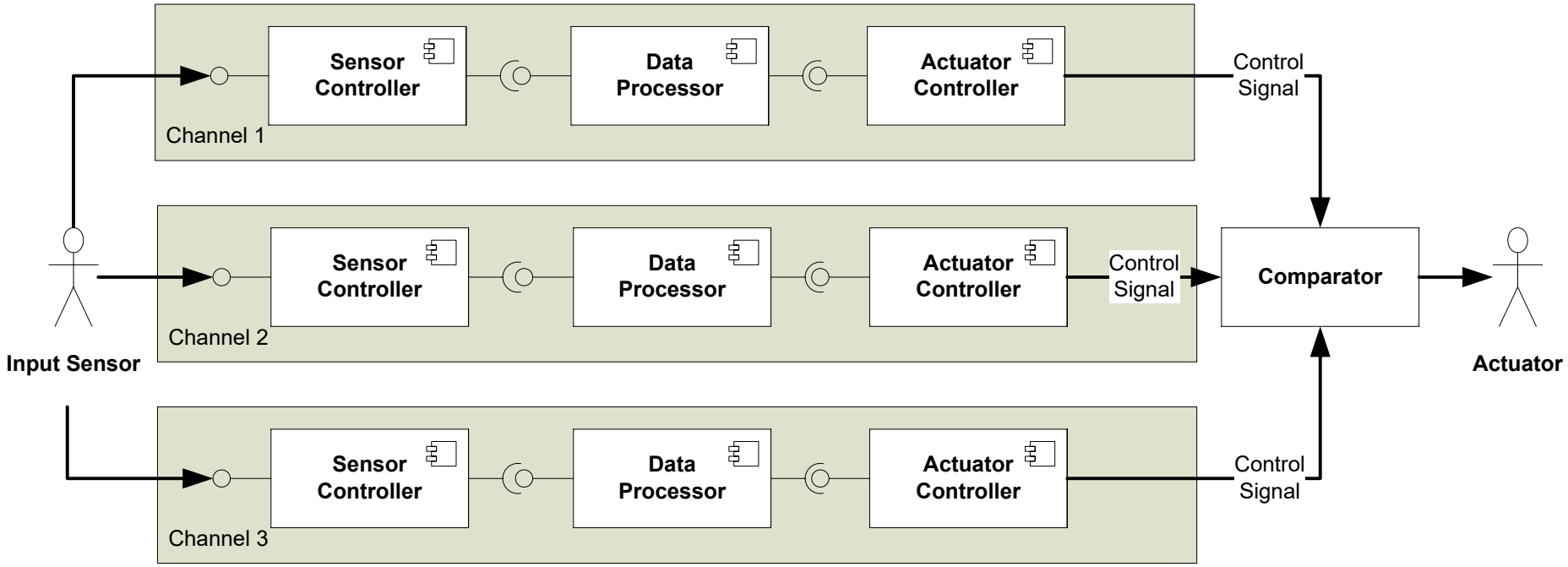


Triple Modular Redundancy

[Douglass 2003, Cha. 9.4]

- Protection against random faults without a failsafe state
  - provides continuity of operation
  - no validation needed

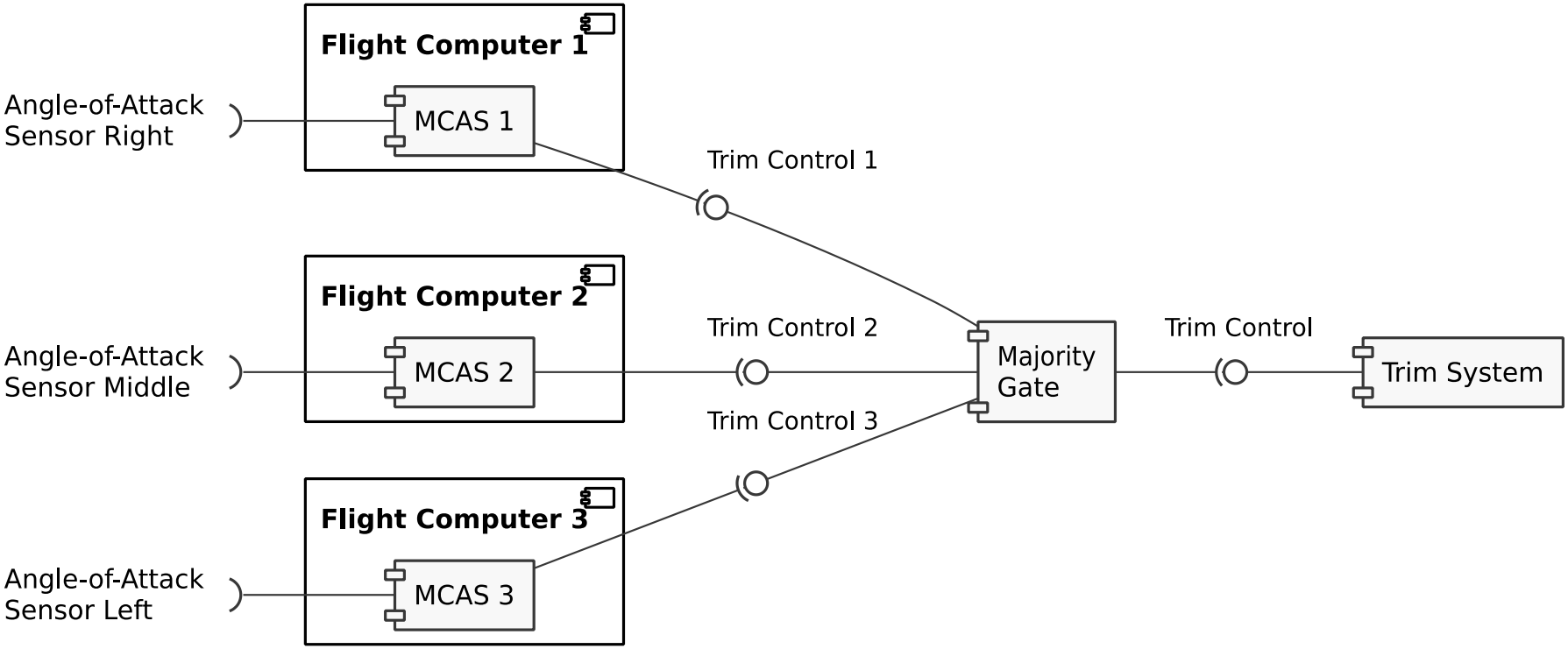
➔ *Does this work to improve pure software solutions?*



Example: Triple Modular Redundancy

- Three flight computers
  - running MCAS simultaneously
  - connected to seperate sensors / actuators
- Control signal determined by majority

By MB-one & Marc Lacoste - This file was derived from: ILA 2018, Schönefeld (1X7A5288).jpg with background clipped, CC BY-SA 4.0

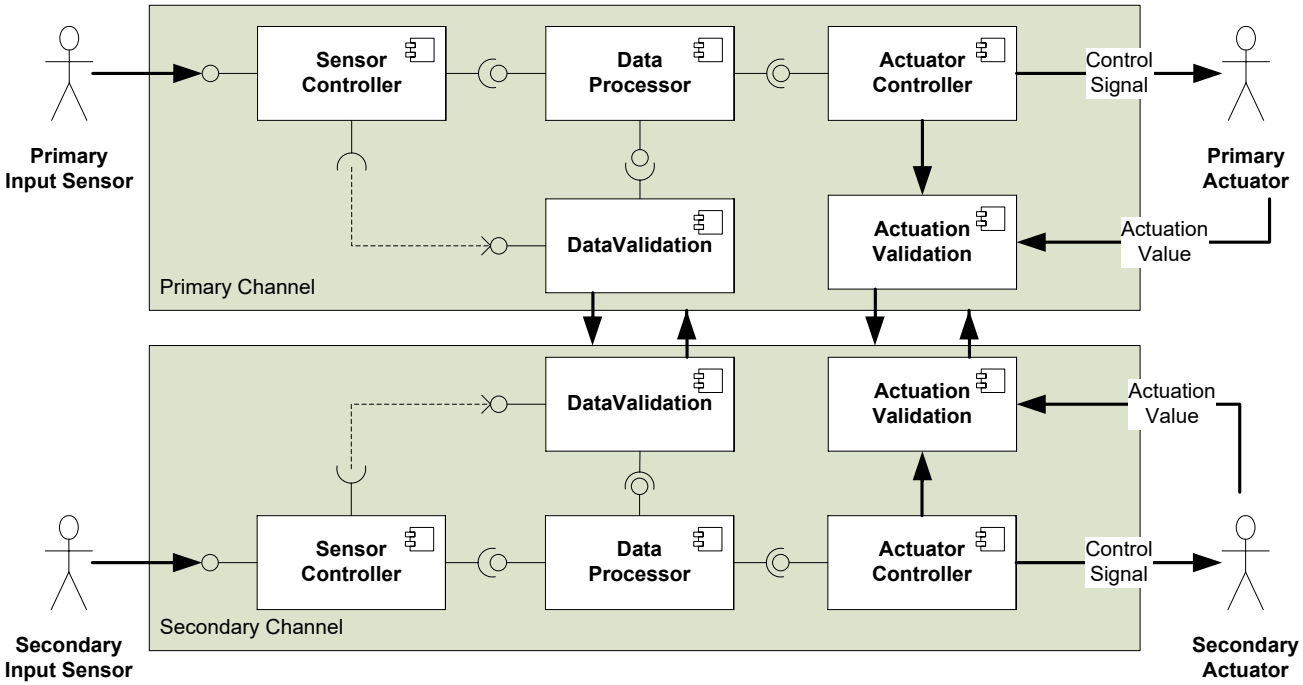


Heterogeneous Redundancy

[Douglass 2003, Cha. 9.5]

- Similar to Homogeneous Redundancy
  - but independent design and implementation of channels required
- Protection against random and systematic faults
  - guarantees continuity of operation
  - without requiring a fail safe state

If not same mistakes are made in both versions…



Safety & Reliability Pattern Overview

[Douglass 2003, Cha. 9]

Without Fail-Safe State

- Homogeneous Redundancy *(Cha. 9.3)*
- Triple Modular Redundancy *(Cha. 9.4)*
- Heterogeneous Redundancy *(Cha. 9.5)*

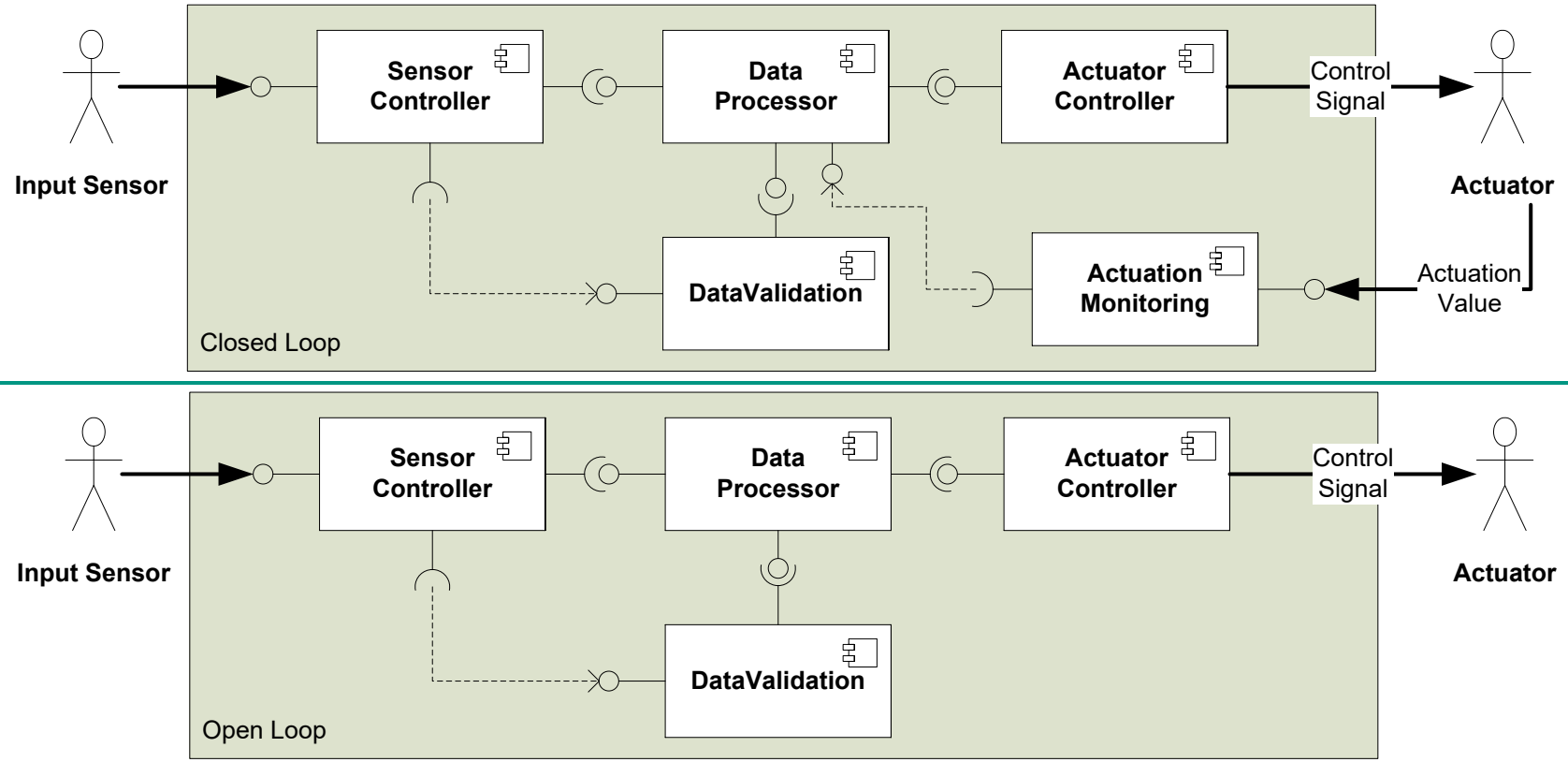
With Fail-Safe State

- Protected Single Channel Pattern *(Cha. 9.2)*
- Monitor-Actuator Pattern *(Cha. 9.6)*
- Sanity Check Pattern *(Cha. 9.7)*
- Watchdog Pattern *(Cha. 9.8)*
- Safety Executive Pattern *(Cha. 9.9)*

Protected Single Channel

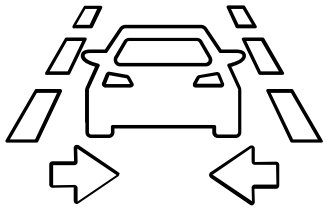
[Douglass 2003, Cha. 9.2]

- Cheaper safety increase without redundancy but through error detection on channel
  - assuming a fail-safe state is available
  - May be able to handle transient faults (e.g. by retry)

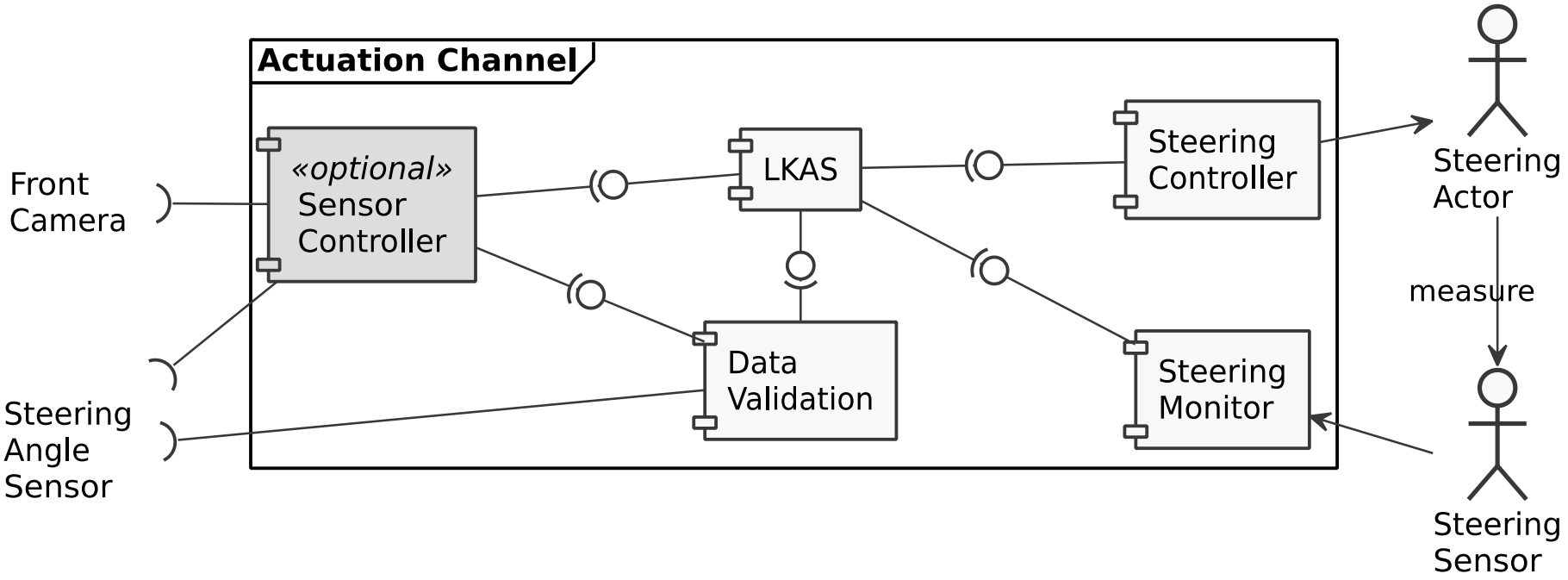


Example: Protected Single Channel

- Validate sensor data and data received by LKAS
- Monitor data send to Steering Controller and measure actual Steering
- Assumes *fail-safe state*
  LKAS turned off in case of failures (driver resumes control)



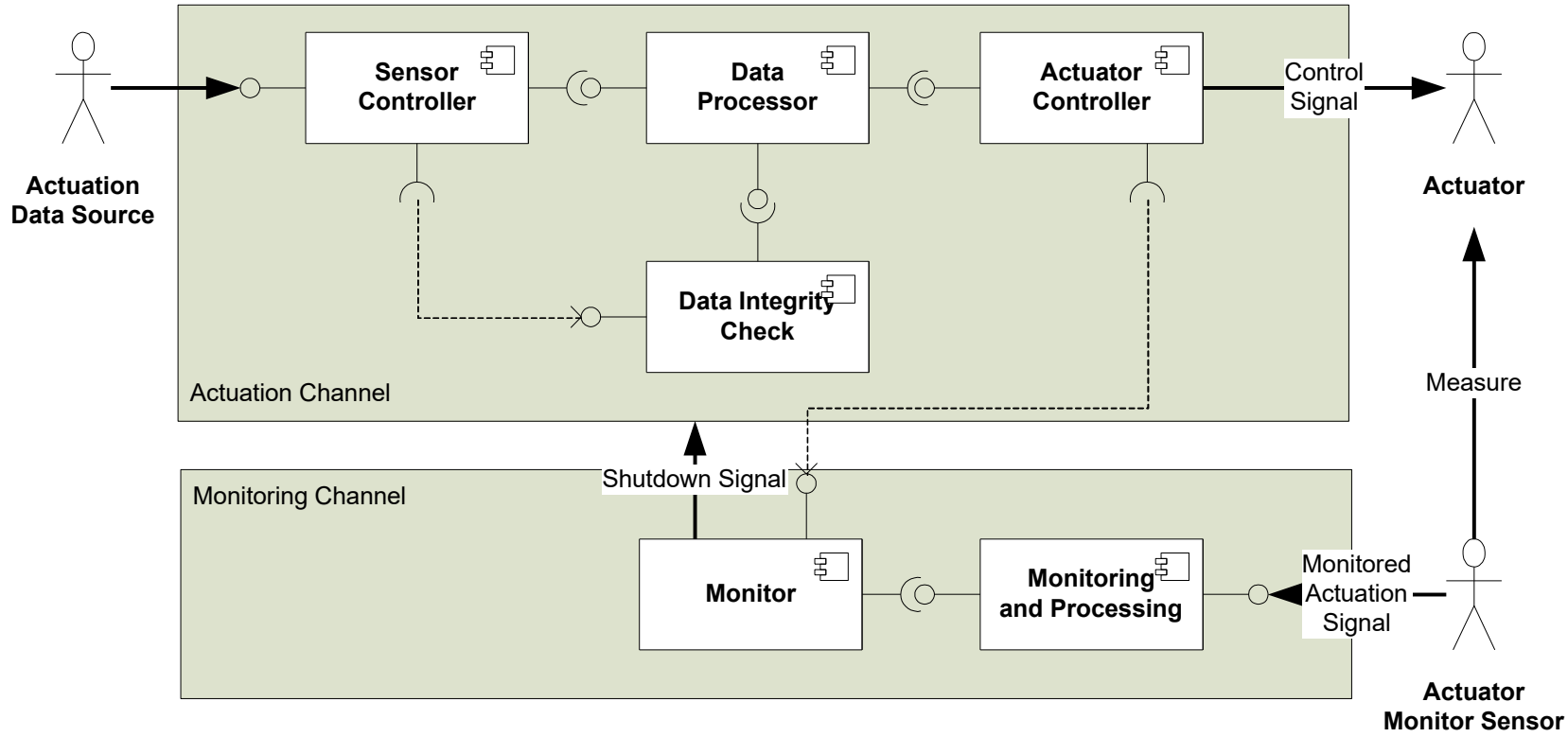
lka car lane keep assist
by Gianluca Pettenon
from the Noun Project
(CC-BY-SA3.0)



Monitor-Actuator Pattern

[Douglass 2003, Cha. 9.6]

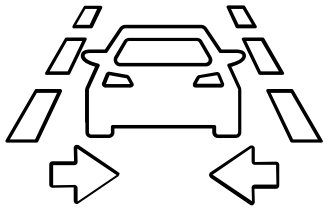
- Protection against random and systematic faults with a fail-safe state
- Special, limited case of heterogeneous redundancy
- Supervision of channel and actuator
- *Actuator Monitor Sensor* must be an independent sensor

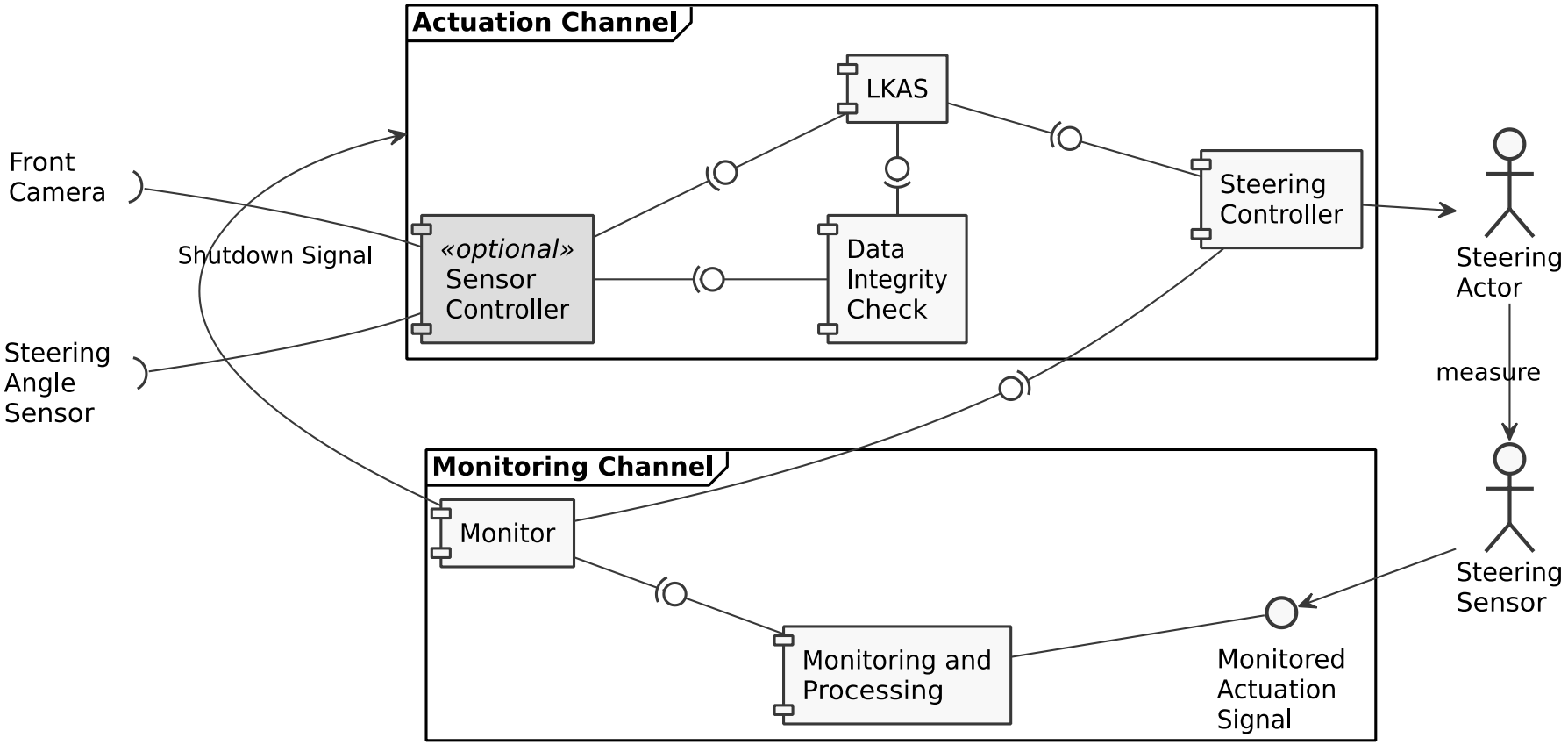


Here, the Actuator Monitor Sensor is mandatory

Example: Monitor-Actuator Pattern

- LKAS with Data Integrity Check in Actuation channel identifying sensor fault
- Monitoring channel identifying actuation fault

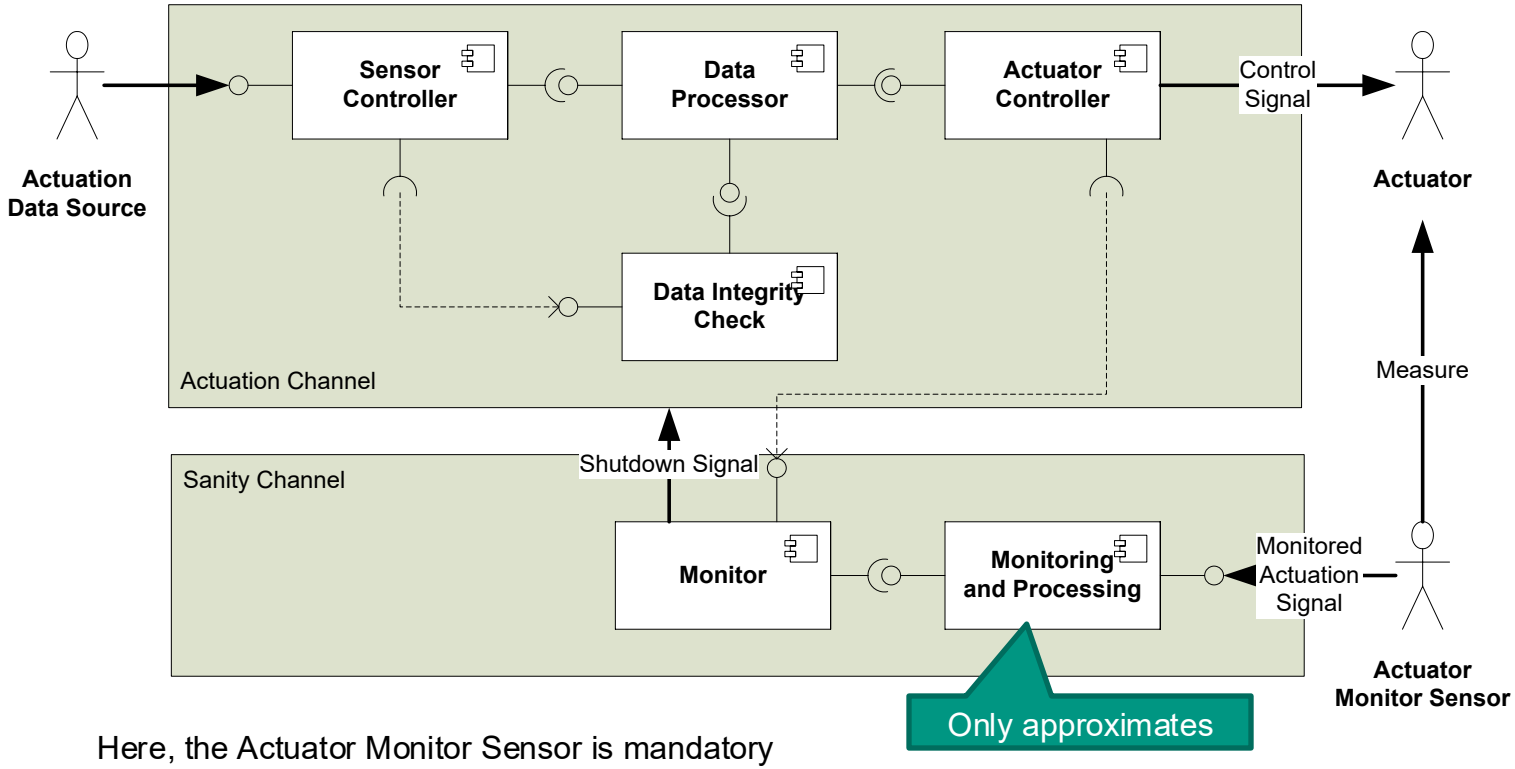




Sanity Check Pattern

[Douglass 2003, Cha. 9.7]

- Lightweight Protection against random and systematic faults with a fail-safe state
  - Derived from Monitor-Actuator pattern
  - Only approximation of results

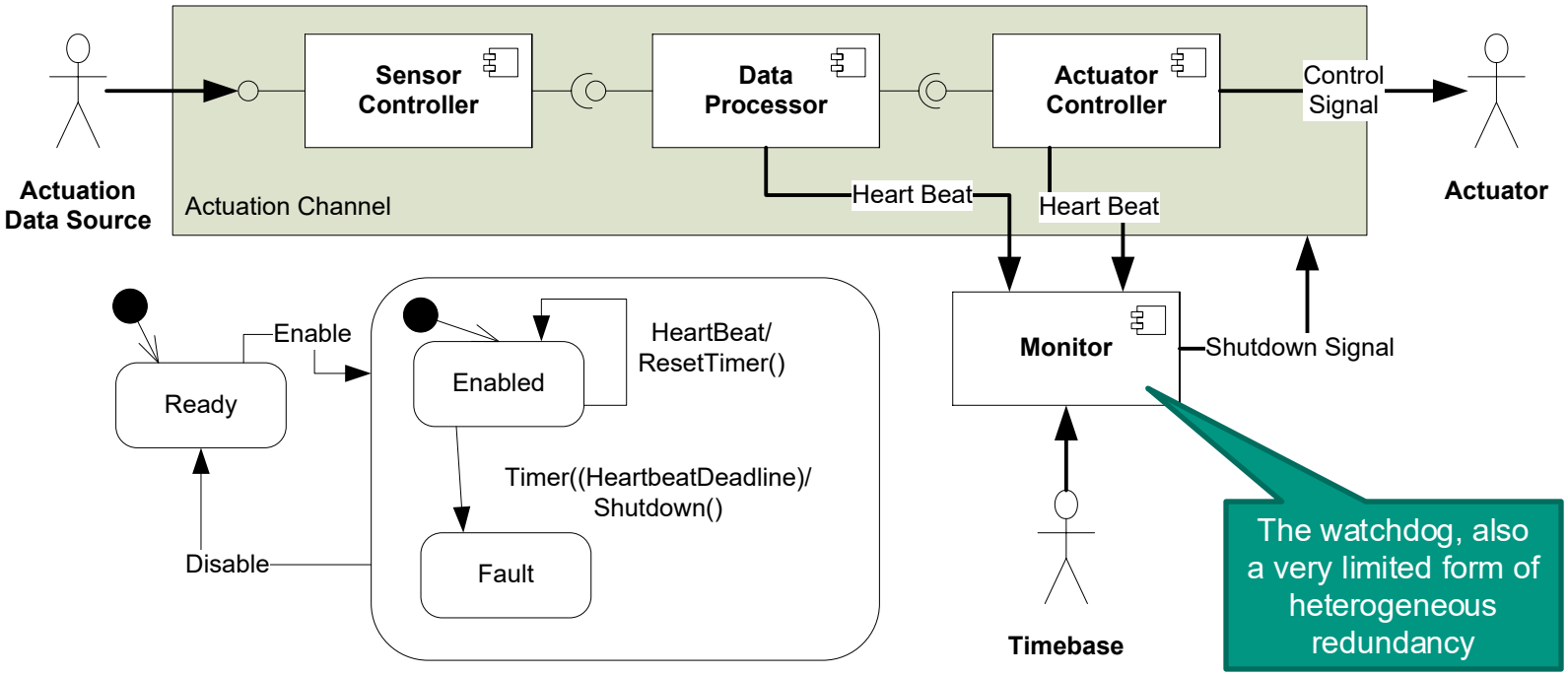


Here, the Actuator Monitor Sensor is mandatory

Watchdog Pattern

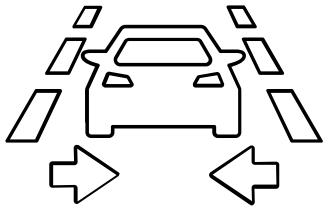
[Douglass 2003, Cha. 9.8]

- Very lightweight protection against time-based faults and detection of deadlocks with a fail-safe state
  - simple error detection on channel, no supervision of actuator
  - channel sends liveness messages to watchdog
    - it is *stroking* it

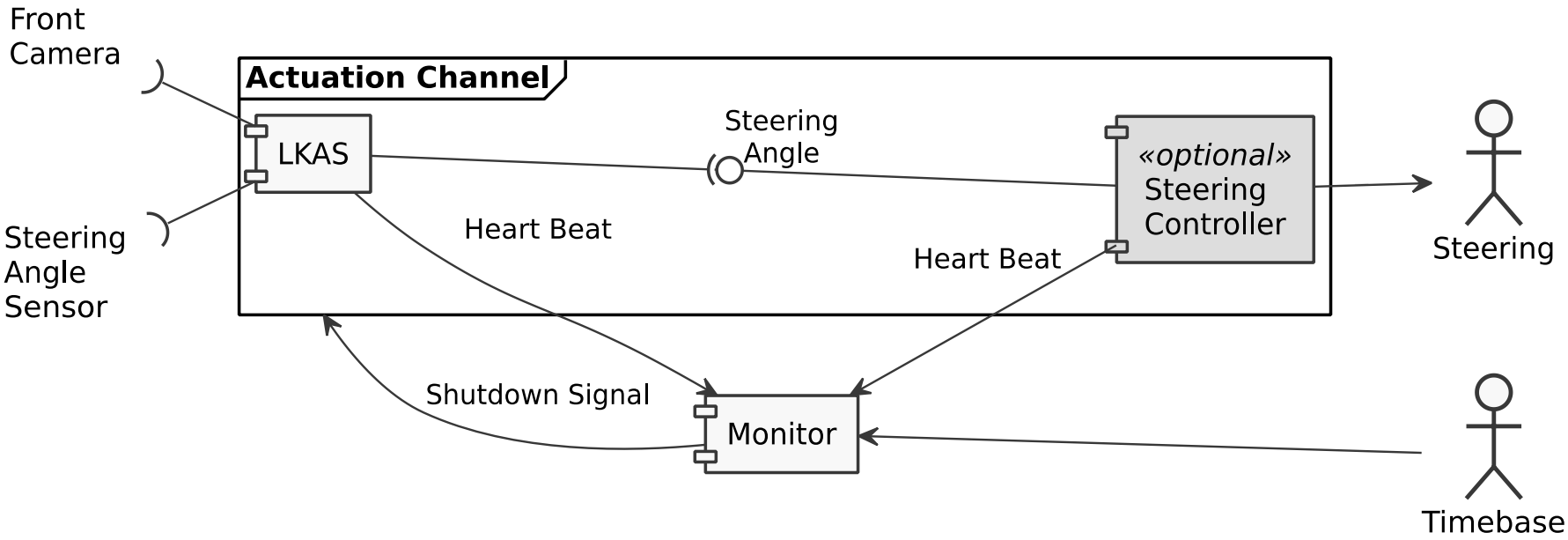


Example: Watchdog Pattern

- LKAS sends heart beat to system monitor
- Monitor continuously checks Heart Beat
- Shutdown Actuation Channel (LKAS), if Heart Beat not present for certain time



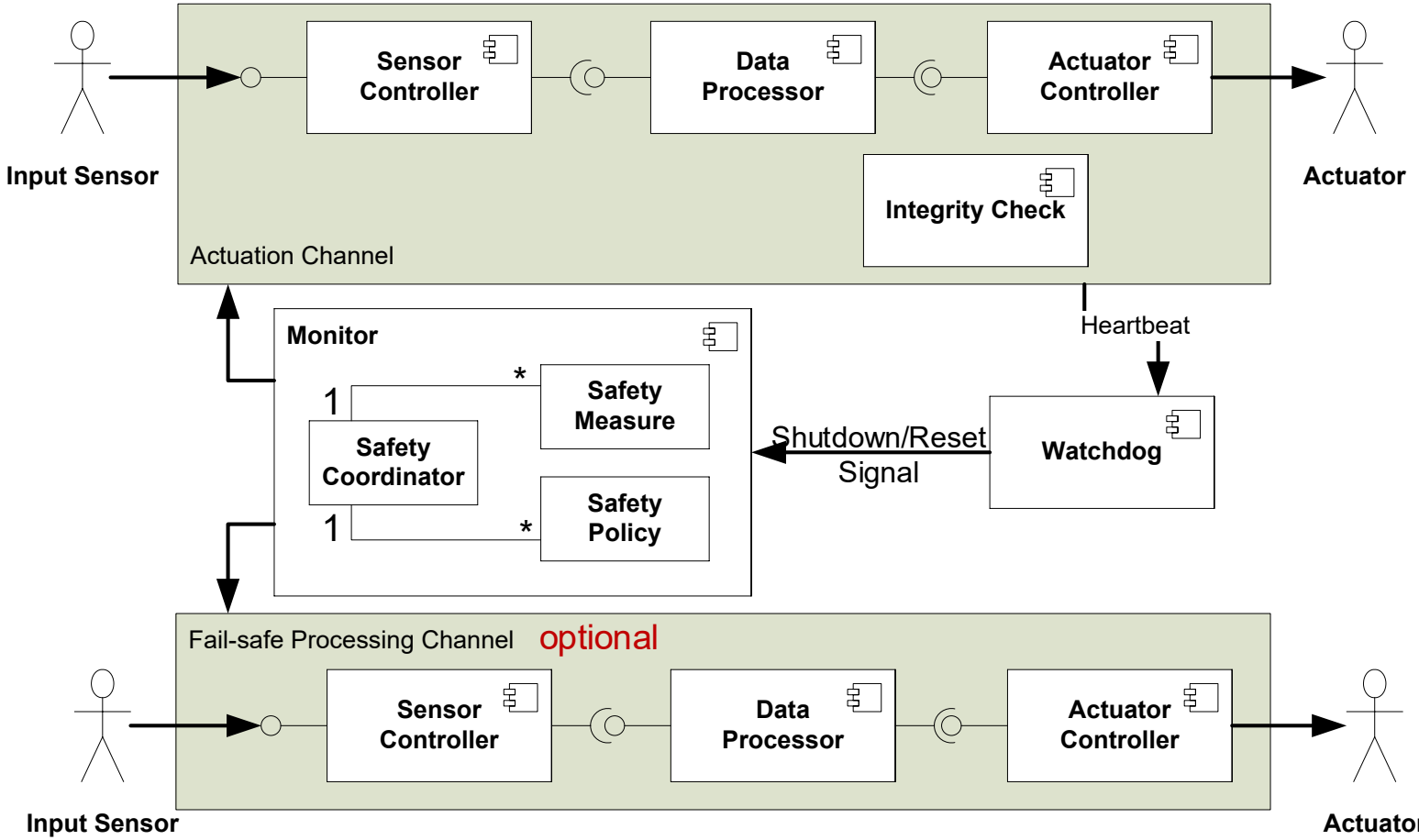
lka car lane keep assist
by Gianluca Pettenon
from the Noun Project
(CC-BY-SA3.0)



Safety Executive Pattern

[Douglass 2003, Cha. 9.9]

- Safety for **complex** systems with **non-trivial** mechanisms to achieve fail-safe state



Safety & Reliability Pattern Summary

[Douglass 2003, Cha. 9]

Without Fail-Safe State

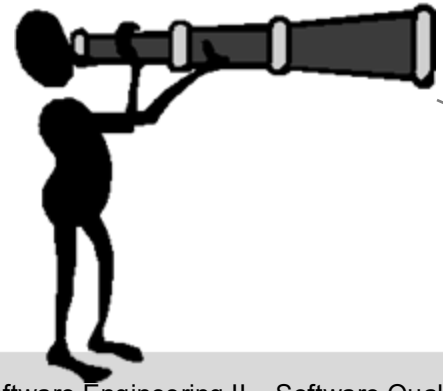
- Homogeneous Redundancy *(Cha. 9.3)*
  Protection against random faults
- Triple Modular Redundancy *(Cha. 9.4)*
  Protection against random fault with continuation of functionality
- Heterogeneous Redundancy *(Cha. 9.5)*
  Protection against random and systematic faults

With Fail-Safe State

- Protected Single Channel Pattern *(Cha. 9.2)*
  Safety without heavyweight redundancy
- Monitor-Actuator Pattern *(Cha. 9.6)*
  Protection against random and systematic faults
- Sanity Check Pattern *(Cha. 9.7)*
  Lightweight protection against random and systematic faults
- Watchdog Pattern *(Cha. 9.8)*
  Lightweight protection against time-based faults and detection of deadlock
- Safety Executive Pattern *(Cha. 9.9)*
  Safety for complex systems with complex mechanisms to achieve fail-safe states, which are more complex to reach

Conclusion

- What have we learned –
  - Real-time system correctness not only depends on what system does, but also on how fast it reacts
    - specific real-time operating systems are available
    - real-time systems architectures designed as number of concurrent processes
  - Monitoring and control systems poll sensors and send control signals to actuators
    - RT system design involves associating processes with sensors and actuators
    - Because of timing demands made by different stimuli/responses every sensor/actor pair should have its own process
  - Safety and reliability in real-time systems
    - Fail-safe states, patterns to achieve safety and reliability



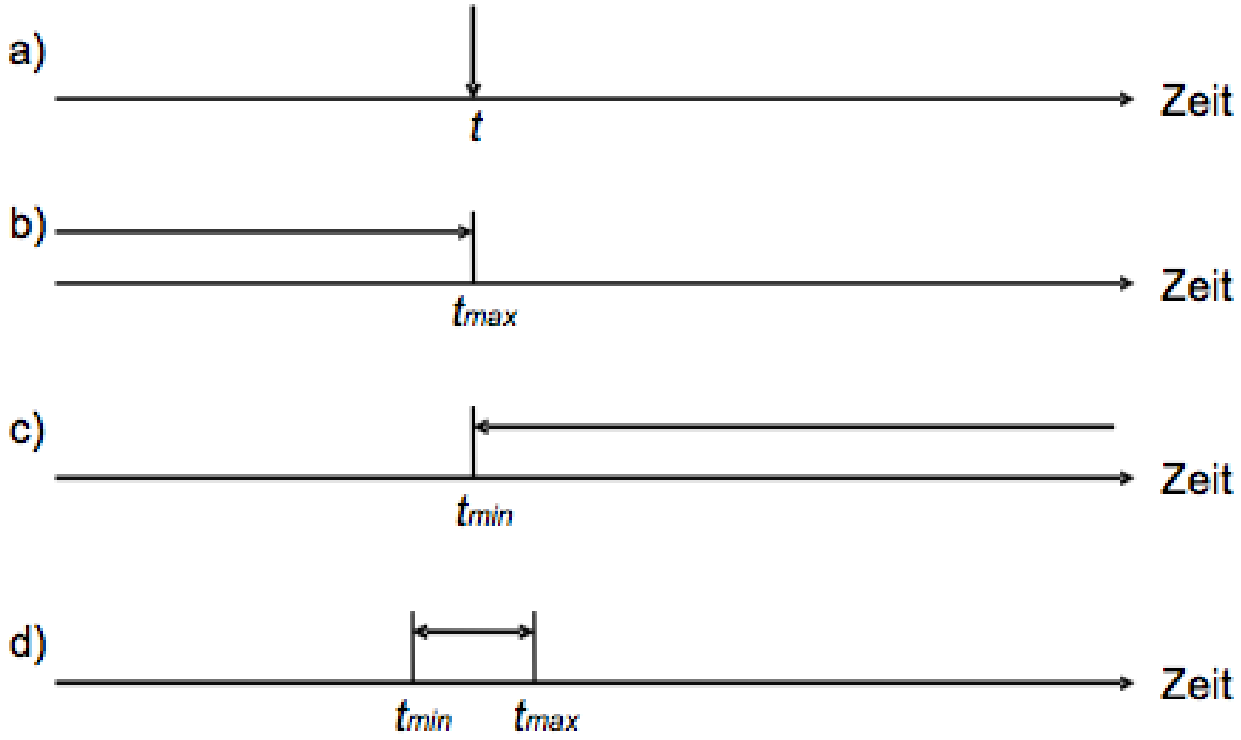
Wrap Up & Colloquium

Literature

- I. Sommerville, Software Engineering,
  - [Sommerville] 7th ed., AW Chapter 15: RT Software Design (unchanged in 8th ed.)
  - [Sommerville 2017] 10th edition, chapter 21
- [Douglass 2003] B. P. Douglass, Real-Time Design Patterns, AW, 2003
- Lecture Echtzeitsysteme
  - SS 2010: Prof. Dr.-Ing. H. Wörn, PD Dr.-Ing. Th. Längle
- [Bucci 1995] Giacomo Bucci, Maurizio Campanai, and Paolo Nesi, Tools for specifying real-time systems. Real-Time Systems, March/May 1995, Volume 8, Issue 2-3, pp 117-172

APPENDIX

Temporal Constraints

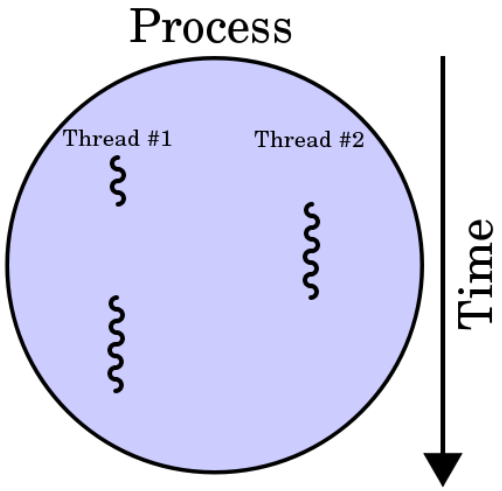


a) Fixed point in time
b) Latest allowed point in time
c) Earliest point in time
d) Timeslot (Jitter)

Terminology: “Thread” vs. “Process”

Usual understanding (from Wikipedia):

- Multiple threads can exist within the same process and share resources such as memory
- Different processes do not share these resources
- The threads of a process share its instructions (the code) and its context (the values of its variables at any given moment).



http://en.wikipedia.org/wiki/Thread_(computing)

- UML modelling: Thread is basic unit of concurrency
  - Lightweight threads (= thread above) if shared memory
  - Heavyweight threads (= process above) if no shared memory
- Sommerville: Only mentions processes as the unit of concurrency

- → „Process“ and „thread“ in this lecture just mean „unit of concurrency“

Properties of a Thread [Douglass, p. 206]

- Thread is basic unit of concurrency in UML
  - Modelled to analyse concurrency
- Arrival pattern can be periodic or aperiodic
  - periodic
    - period: time between invocations
      - jitter („Flimmern“): potential variation in the period
    - process deadline - the time by which processing must be complete
  - aperiodic
    - minimum inter-arrival time
    - maximum burst length: maximum number of arbitrarily close thread invocations
    - ➔ *probability distribution of thread invocations required for modelling*
- Execution time
  - worst case
    - depending on logic per thread invocation
  - average
  - ➔ modelling usually requires probability distribution of execution times
- The above is simple case, assumes independence of threads

Thread Communication I

- Communication can be done via –
  - memory
    - shared variables
  - messaging
    - synchronous or asynchronous
    - blocking or non-blocking
- Communication via shared variables (memory communication)
  - the alternative message-based communication is usually slow(er)
  - requires access control to ensure right ordering between read- and write-accesses to avoid typical errors –
    - reading out-dated values
      - wrong order of read and write
    - overwriting data
      - two writes without read in between
    - creation of inconsistent data
      - e.g., with non-atomic increment

Thread Communication II

- Communication via messages
- Behaviour of sending operation
  - blocking
  - non-blocking
    - need to check for completion
    - message IDs required if multiple outstanding sending requests can occur
- State of receiver during communication
  - synchronous
    - sender and receiver reached their respective communication operation
  - asynchronous
    - receiver may not have reached the receiving operation
      - ➔ buffering of messages required

Petri Nets in a Nutshell

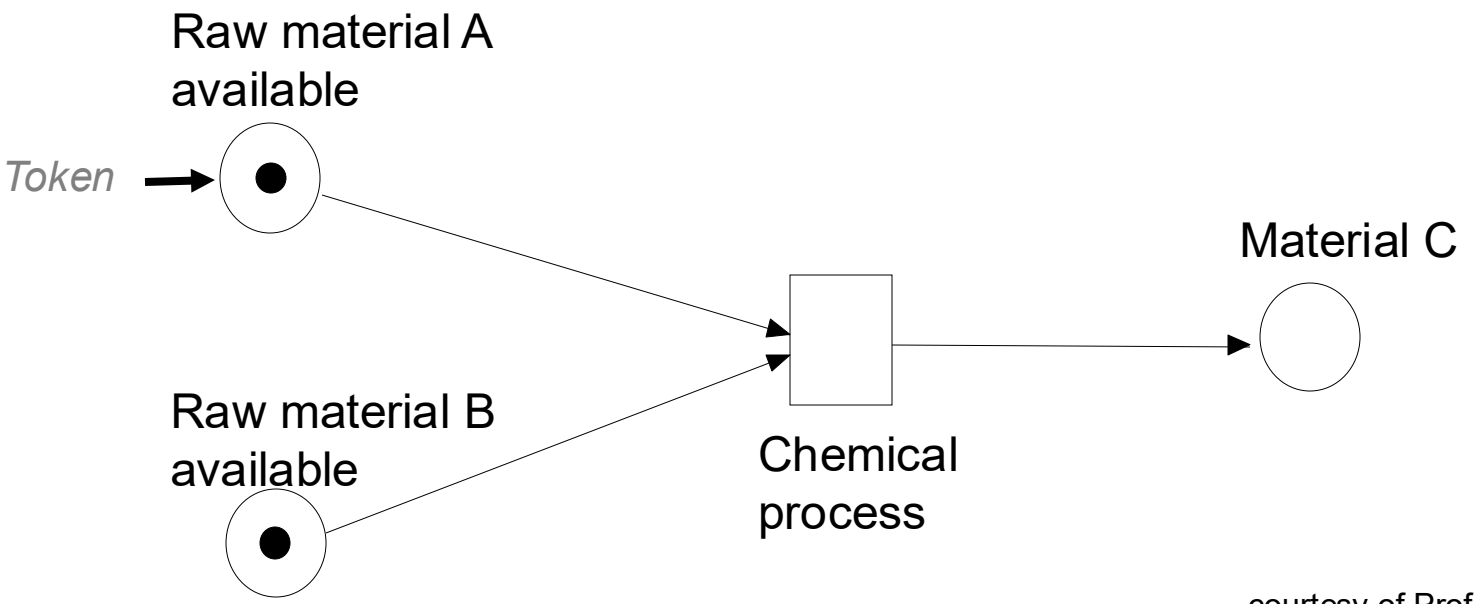
- Concurrency has traditionally been modelled with Petri Nets
  - developed by Carl Adam Petri in 1962
  - based on graph theory and thus have precise semantics
    - ideally suited to analyse concurrent processes, in order to e.g. detect deadlocks
  - Extensions such as Timed Petri Nets or Stochastic Petri Nets to analyse timing properties

- Basic elements –
  - places 
    - describe the current states (resp. conditions) of a system
  - transitions 
    - stand for events (resp. actions) that transition the system in a new state
  - arcs 
    - connect places and transitions

Background (optional) on Tools for analysing real-time systems: [Bucci 1995]

Petri Net Example

- For graphical analysis Petri Nets use tokens
  - in the simplest case, each place may carry one token
    - a token in a place means its condition is true
    - unmarked places represent unmarked conditions
- A transition may fire as soon as all its incoming places are filled
  - and the outgoing place is empty



courtesy of Prof. Dadam, Uni Ulm

RT System Analysis

- Petri nets can be used to analyse system flow
  - i.e. can race conditions and deadlocks occur?
- However, it is also important to analyse adherence to timing constraints
  - static analysis techniques exist for this purpose
    - not a trivial task
    - typically requires probability distributions for the occurrence of aperiodic events

➔ It may uncover that the system is not able to keep its timing constraints

- potentially some functions need to be implemented in hardware in this case

- *Modern information systems may also have „deadlines“*
  - *e.g. SLAs in cloud-based systems → can e.g. be analysed with Palladio*

UML Concurrency Modeling

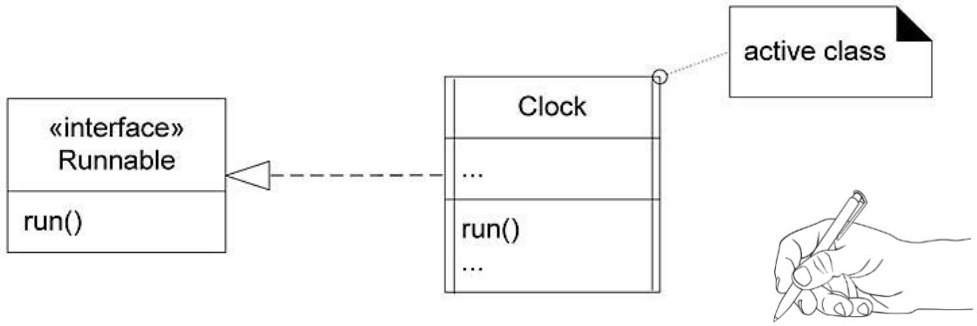
“Indeed, building a large piece of software is hard enough; designing one that encompasses multiple threads of control is much harder because one must worry about such issues as dead-locks, livelock, starvation, mutual exclusion, and race conditions.” -- Grady Booch

- In the UML, threads resp. concurrency can be modelled as follows –
  - ……
  - ……
  - ……
  - ……

UML Concurrency Modeling

“Indeed, building a large piece of software is hard enough; designing one that encompasses multiple threads of control is much harder because one must worry about such issues as dead-locks, livelock, starvation, mutual exclusion, and race conditions.” -- Grady Booch

- In the UML, threads resp. concurrency can be modelled as follows –
  - .....
  - .....
  - .....
  - .....



Concurrency Patterns [Douglass, Chap. 5]

Communication:

- Message Queuing Pattern: Robust asynchronous task communication (queue protected by mutex)
- Guarded Calls Pattern: Robust synchronous task communication (communication via a shared mutex-proteced ressource)
- Rendezvous pattern: generalised task rendezvous (arbitrary preconditions for communications), generalisation of Guarded Calls Pattern
- Interrupt pattern: Fast short event handling

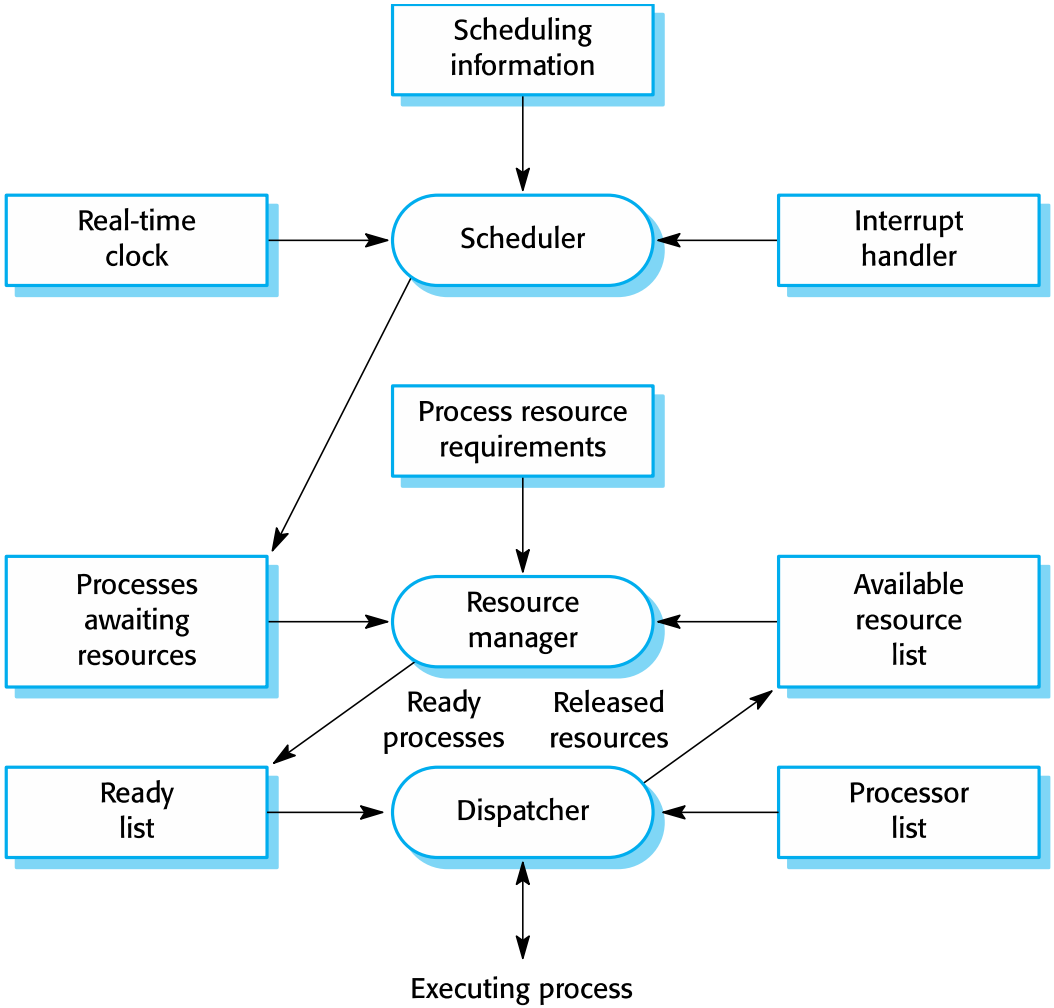
Real-Time Operating Systems

- Common OSs usually carry a lot of overhead like –
  - file & network management, UI support etc.

  … and are thus not well suited for RT operations
- Real-time operating systems are specialised for real-time systems
  - and usually include the following components needed for them –

real-time clock	provides information for process scheduling
interrupt handler	manages aperiodic requests for service
scheduler	chooses the next process to be run
resource manager	allocates memory and processor resources
dispatcher	starts process execution

RTOS Overview



[Sommerville]

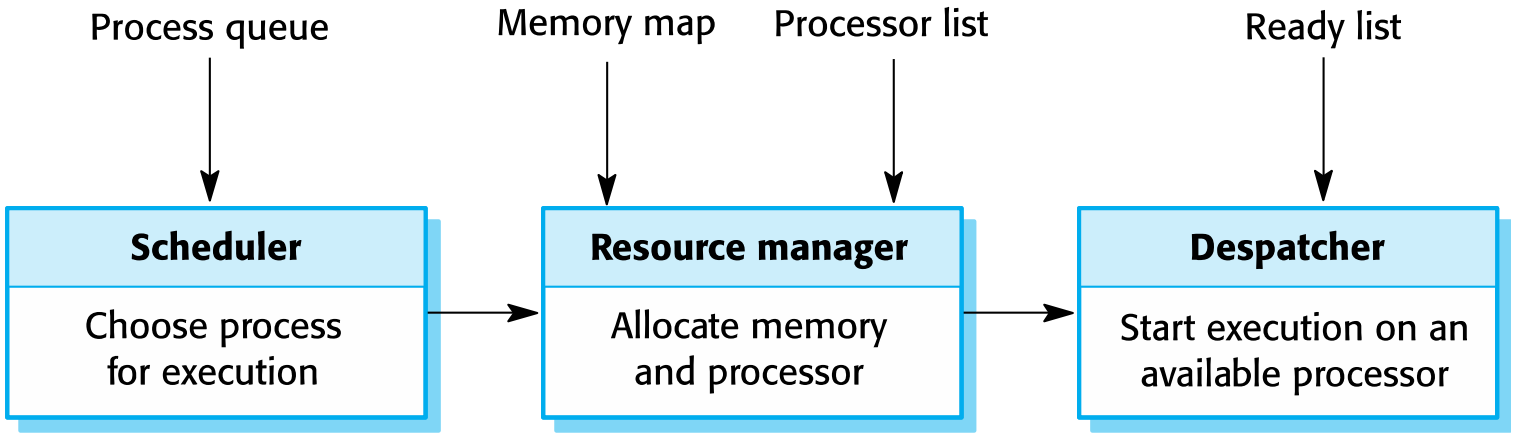
Process Management

- Concerned with managing set of concurrent processes
  - periodic processes are executed at pre-specified time intervals
- RTOS scheduler uses real-time clock to determine when to execute process taking into account the –
  - arrival pattern
  - execution times
  - process priority
- RT systems must be able to support at least two priority levels –
  - Interrupt level priority
    - highest priority, which is allocated to processes requiring immediate response
  - Clock level priority
    - allocated to periodic processes

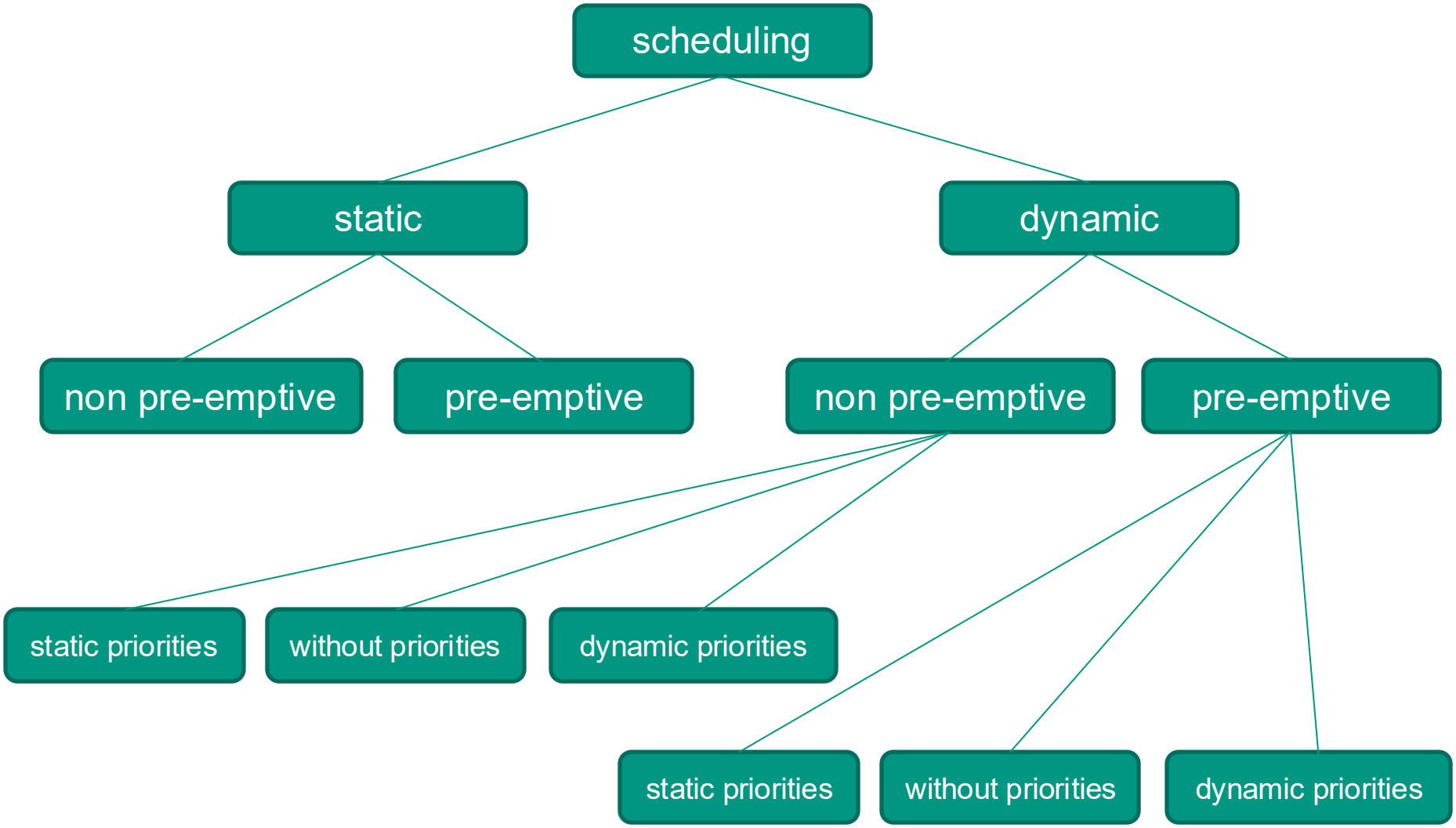
  ➔ within these, further priority classes may be assigned
    - *e.g., for faster vs. slower devices*

(Periodic) Process Management

1. Scheduler chooses next process to be executed by processor
   - this depends on scheduling strategy, which may considers process priority
2. Resource manager allocates memory and processor for process to be executed
3. Dispatcher takes process from ready list, loads it onto processor, and starts execution



Classification of Scheduling Strategies



Some Scheduling Strategies

- FIFO – First In First Out
  - dynamic, non pre-emptive, without priorities
- Fixed Priorities
  - dynamic, static priorities
- Earliest-Deadline-First-Scheduling (EDF)
  - dynamic, dynamic priorities
- Least-Laxity-First-Scheduling (LLF)
  - dynamic, dynamic priorities
  - now + laxity + remaining processing time = deadline
  - ➔ laxity = deadline – now – remaining processing time
    - ➔ negative laxity means that task cannot make its deadline anymore
- Time-Slice-Scheduling
  - dynamic, pre-emptive, without priorities

Java as Real-Time Language?

- Hard real-time systems may have to be programmed in assembly language
  - to ensure that deadlines are met
  - this trend lowers considerably, due to –
    - bad portability of assembler code and low productivity
    - and because languages such as C also allow efficient programs to be written
      - however, they lack constructs to support concurrency or shared resource management without additional libraries (such as threads)
- Standard Java offers built-in constructs for concurrency, but has problems with real-time execution due to –
  - when should a thread be started?
  - automated garbage collection interrupting threads
  - dynamic class loading and linking
  - JIT (Just-in-Time Compiling)
  - encapsulation of hardware
  - ➔ however, specific Java extensions exist, *e.g., JAVA/RT*

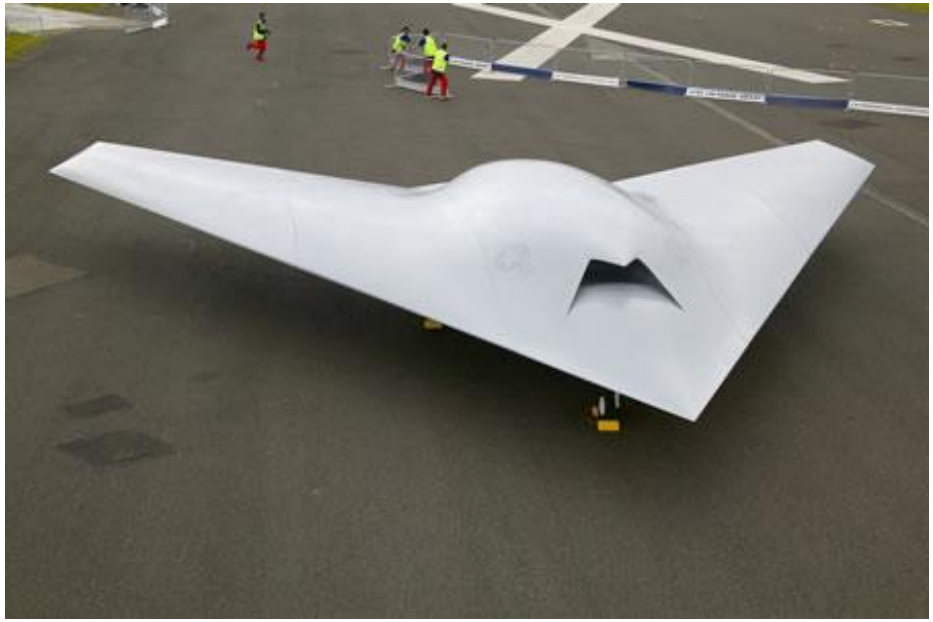
JamaicaVM: Real-Time JVM from Karlsruhe

- Hard Real-time Execution
  - including features essential to object-oriented software development
    - like dynamic allocation of objects, inheritance, and dynamic binding
- Real-time Garbage Collection
  - GC operates in small increments of only a few machine instructions, but still guarantees –
    - to recycle all garbage memory
    - to avoid memory fragmentation
    - to limit execution time for allocations
- Extensive tool suite for build process
  - www.aicas.com
    - free evaluation version for download

➔ *Founded by graduates of the IPD*

Example for Java in RT

- X-45C UCAS
  - Unmanned Combat Air Systems
  - Aonix PERC
    Java J2SE-based real-time embedded virtual machine



http://www.boeing.com